BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 3339

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT No. 6
1 April 1976 to 30 June 1976

and

FINAL REPORT
CONTRACT LINE ITEM 0001AC
PLURIBUS MESSAGE SYSTEM STUDY

Principal Investigator: Mr. Frank E. Heart
Telephone (617) 491-1850, Ext. 470

Sponsored by:
Advanced Research Projects Agency
ARPA Order No. 2351, Amendment 15
Program Element Codes 62301E, 62706E, 62708E

Contract No. F08606-75-C-0032
Effective Date: 1 January 1975
Expiration Date: 1 April 1977
Contract Amount: $2,655,355

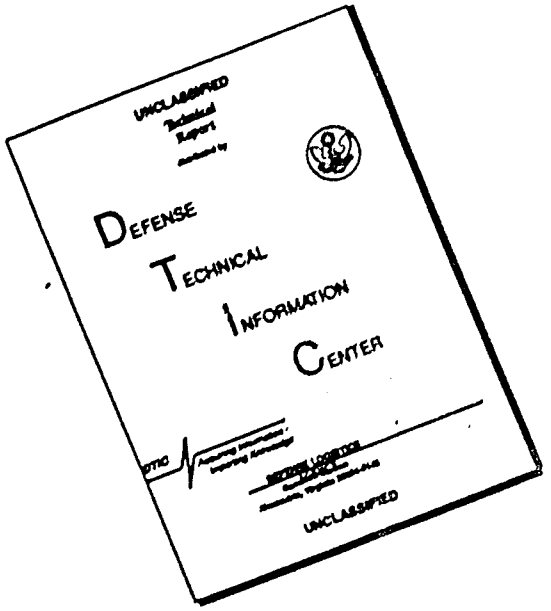
Title of Work: Operation and Maintenance of the ARPANET

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Report No. 3339 Bolt Beranek and Newman Inc.

INTERFACE MESSAGE PROCESSORS FOR THE ARPA COMPUTER NETWORK

QUARTERLY TECHNICAL REPORT NO. 6
1 Apr [redacted] o-30 Jun 1976

and

FINAL REPORT
CONTRACT LINE ITEM 0001AC
PLURIBUS MESSAGE SYSTEM STUDY

Submitted to:

IMP Program Manager
Range Measurements Lab.
Building 981
Patrick Air Force Base
Cocoa Beach, Florida 32925

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of the Advanced Research Projects Agency or the U.S. Government.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE of TABLES

1. INTRODUCTION

This Quarterly Technical Report, Number 6, describes aspects of our work performed under Contract No. F08606-75-C-0032 during the second quarter of 1976. The first four reports in this series dealt largely with work quite closely related to the development, maintenance, and operation of the ARPANET, e.g., the IMPs and TIPs of the ARPANET and the Satellite IMPs and PLIs connected to the ARPANET. However, beginning with last quarter and continuing this quarter, our work with the ARPANET has been funded under a contract from the Defense Communications Agency and our work with Satellite IMPs, PLIs, etc. is being reported elsewhere. The only significant body of work still funded under this contract is a study of the feasibility of using the Pluribus computer as the basis of a large, secure message system. In fact, even this body of work will be complete very early in the third quarter. Thus, the remainder of this document describes our study of a Pluribus Message System (PMS) and is a final report on this work, in accordance with Contract Line Item 0001AC[1].

[1] Quarterly Technical Report 5 in this series presented an interim report on this work.

1.1 Overview of the Study

Message handling systems have been developed on several hosts on the ARPANET and have received widespread use. We have studied an extension of this message technology based on the Pluribus computer line, contemplating a system which will provide a message handling service of high capacity and high reliability while meeting necessary security requirements.

In many areas, the natural advantages of the Pluribus make it extremely attractive as the base of a message system.

- Because the Pluribus has a highly modular architecture, it has the flexibility to handle a wide range of configurations.

- Because the Pluribus is a multiprocessor, it has the processing capacity to handle very large systems.

These first two features make the Pluribus one of the most flexible computer systems available today. This is important, since a message switching system should be matched to its user base and should be able to expand gracefully as the user base grows.

- The Pluribus offers high system availability.

Message systems, particularly large ones with many users, are distinguished by their need for high system availability. The Pluribus is particularly appropriate because it meets this need at a low cost. Reliability is achieved by providing at least one spare copy of every system resource. For example, if a Pluribus with 10 processors is required to serve a given application, an eleventh processor would be added to take over should any one processor fail. This is much less expensive than duplicating the entire system. This same philosophy is applied to all levels of the system, i.e., memory and I/O as well as processors.

- The novel Pluribus architecture is available off the shelf.

Several Pluribus systems have already been delivered and are presently providing their users with reliability and flexibility.

The advantages outlined above make the Pluribus highly attractive as a message system. However, the Pluribus is a novel machine which was developed for a specific application; as a result, it lacks some of the features provided with most large mainframes (e.g., mass storage, software for a file system, an operating system, etc.) These features are not technically difficult to provide on the Pluribus -- they were simply not

required for previous applications and thus were never developed. We have investigated how best to add such features to the Pluribus and have described our findings in this report. We believe that the effort and cost of augmenting the Pluribus in these ways is more than offset by the advantages of flexibility, reliability, and available computing power.

In studying the design of a Pluribus-based message handling system, we considered the developmental effort required in the following five areas: 1) Mass Storage devices (Disks); 2) file system; 3) message system recovery techniques; 4) security techniques; and 5) high order language compilers. These five developmental areas are introduced, in turn, in the following five paragraphs.

The disk which is currently supported on the Pluribus is not large enough to meet the storage requirements of a message system. Section 2.4 of this report describes how IBM-3330 compatible disk units can readily be used to supply the necessary mass storage capability. In addition to disk units, electronic beam addressable memories (EBAMs) seem to be very attractive for a large message system application because of their short access time, e.g., for swapping memory.

A file system which stores messages and keeps track of them is described in section 3.1. It makes efficient use of the disk space by storing only one copy of each message on the disk, regardless of the number of "owners" or "copies" of the message. It manipulates messages by maintaining lists of pointers to them instead of actually moving the messages. This both saves processing time and reduces the required disk space by almost a factor of three. The remainder of Section 3 discusses other software issues related to the design of a Pluribus message system.

Pluribus software has been implemented which automatically restores system operation when some hardware element breaks. The current Pluribus recovery techniques and new techniques for achieving even higher availability through distributed processing are described in Section 4. New software must be written to restore the state of the disk and the file system to the point before the failure. The recovery techniques use "checkpoints" and "before images" to restore the state of the disk efficiently, and full and incremental dumps are used to recover from gross disk system failures. These techniques are also described in Section 4.

Two other requirements of a message system are privacy and, in some environments, security. Once again, the Pluribus was not designed with security in mind (few computer systems are). However, as presented in detail in Section 5 of this report, we have designed techniques which will enable us to ensure secure operation of a Pluribus message system. These techniques are a novel development of this study, and we believe represent a particularly noteworthy achievement. They permit secure operation at a reasonable cost even in the face of hardware failures. The techniques we recommend are physical separation of the primary message processing functions from the security kernel functions, redundant execution of the security kernel functions in physically separate machines, and checksums and access lists on every important data element in the primary machine. These combine to detect a hardware failure anywhere in the system and thereby assure secure operation. Although these techniques are not themselves new, what is novel is the way we have combined them in a cost-effective manner so as to achieve the requisite level of security.

As a final requirement, there is currently no compiler which generates code for the Pluribus. With such a compiler the software for a Pluribus message system could be written in a High

Order Language (HOL), making the program easier to write, to change, to maintain over its life cycle, and to certify for secure operation. Section 6 discusses the issues involved in choosing among the potentially available HOLs (BCPL, BLISS, and PASCAL and its derivative EUCLID seem to be the best candidates). If we had to choose among them now, we would select BCPL. However, EUCLID is still changing, and it may well become more attractive with time and should be considered carefully before an implementation starts.

Prior to the detailed discussions of the five development areas in Sections 2.4, 3, 4, 5, and 6, the structural alternatives which help "drive" the message system design are discussed in Sections 2.1 to 2.3. The Pluribus configurations we have arrived at are summarized in the section immediately below, Section 1.2. Following the detailed discussions of the development areas in Sections 2.4, 3, 4, 5, and 6, some example systems are sized in Section 7. Two appendices support the main body of the report. In particular, if the reader is not already familiar with the basic Pluribus architecture, Appendix B should be read before reading of the main body of the report.

1.2 Pluribus Message System (PMS) Description

We now describe a Pluribus configuration capable of supporting 1000 active users, a configuration based on detailed calculations presented in Section 7. This configuration provides all the services presented in the body of this report, i.e., it uniformly opts for user service or convenience without excessive regard for cost. (The report makes clear that a rich selection of less potent systems could be selected at lower cost.) The Pluribus hardware required for this system is:

- 17 Processors
- 10 Processor Busses
- 2 300-megabyte Disk Drives
- 2 Tape Drives
- 400,000 Words of Memory
- 3 Memory I/O Busses
- Doubled Interfaces to the Disks, Tapes and Network
- 1 4-megaword EBAM

As shown in Figure 1, this version of the Pluribus Message System has a number of functional units which are integrated to form the total system. For convenience we have named the major parts of the system with the colors orange, blue and green. The main

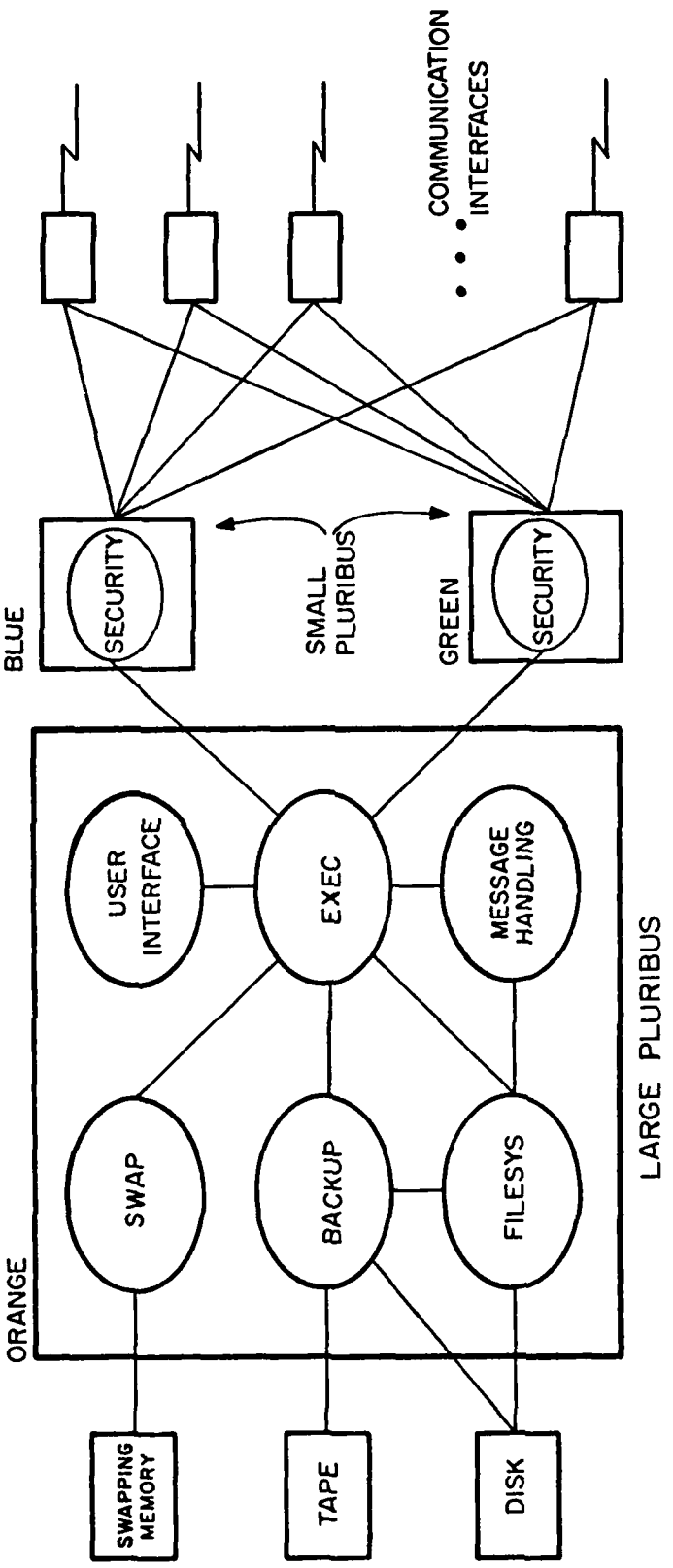


Figure 1 The Pluribus Message System

system is denoted as "orange" and contains an executive which manages all the computer resources and controls the other processes. There is a user interface module and a message handling module which deal with user commands and responses and with message storage and retrieval respectively. The latter calls on the file system and backup system routines for accessing the disks and for archiving all files on tape in case of disk failure. Finally, there are swapping routines which decide which programs are in primary memory and which are in secondary memory at any given time. The smaller "blue" and "green" systems each run critical security code in physically separate small Pluribus computers[2].

ARPA may feel that the proper approach would be to build a smaller, cheaper prototype first. A reasonable configuration for a prototype system would be:

- 7 Processors
- 5 Processor Busses
- 100 megabytes of Disk Memory
- 1 Multi-Line Controller
- 1 Tape Drive

---

[2] This will be elaborated upon in Section 5.

- 250,000 Words of Memory
- 2 Memory I/O Busses
- Doubled Interface to the Network
- Doubled Interface to the Tapes and Disk

This configuration would support 100 users, would have the full functionality of larger system, and would not require an EBAM, the only bit of currently unproven technology in the larger system. Were security required, 3 processors would have to be added. Again, the detailed calculations resulting in these configurations are given in Section 7.

1.3 Conclusions of the Study

As we discuss in the body of this report, we have found that a Pluribus message system is feasible. A message system can be built which is capable of secure operation even in the event of hardware or software failures. Furthermore, this protection will be operational when any module of the system is out of service. We have the following conclusions to report about the Pluribus message system:

- It is responsive. As configured and with 1000 active users, it has adequate capacity to give quick response to user requests for service.

- It is secure. No single hardware failure or software error can result in any message or message fragment being made available to any user other than one who is entitled to see it.

- It is reliable. If a component such as a processor or a memory unit or an I/O interface fails, the software will detect the failure and reconfigure so as to operate without the defective component, without human intervention and in a few tens of seconds.

- Its file system is reliable. All information placed into the system is backed up onto magnetic tape. Even a catastrophic failure of the disk can be recovered from without significant loss of information. Lesser failures will be recovered from more rapidly and with no loss of data.

- It is available. The system's ability to operate with pieces having failed or removed for repair means it is not likely to be unavailable for use when needed.

- It is modular. Should the loading on the system increase, either because of more users or because of a change in the habits of its users, it is easy to add more processors

and/or more memory. Further, at worst only trivial software modifications are required to support such increased hardware.

- It is flexible. Since the system is to be programmed in a High Order Language, it will be easy to modify in order to meet new or changing needs of the users.

We recommend that ARPA seriously consider supporting the development of the Pluribus message handling system we have designed: it has a unique set of benefits and advantages.

2. OVERALL SYSTEM DESIGN

Certain basic questions must be answered before design can start on a Pluribus message system (PMS).

- How will users gain access to the PMS? That is, will they come through a network (such as the ARPANET) or be connected directly to the PMS?
- To what extent will users be able to specify the characteristics of the user interface?
- How (if at all) will the PMS be interfaced to a network?
- What form of mass storage will be used?

These questions are addressed in the following sections.

2.1 Terminal Access

An important question to be answered concerns the way that the users of the PMS obtain access to ths system. Four possible answers to this question are illustrated in Figure 2. The first and simplest, from the point of view of a PMS, is to configure the message system without any local users at all and to rely on some network or other external entity to provide the support for the terminals. This is attractive in that it keeps the message

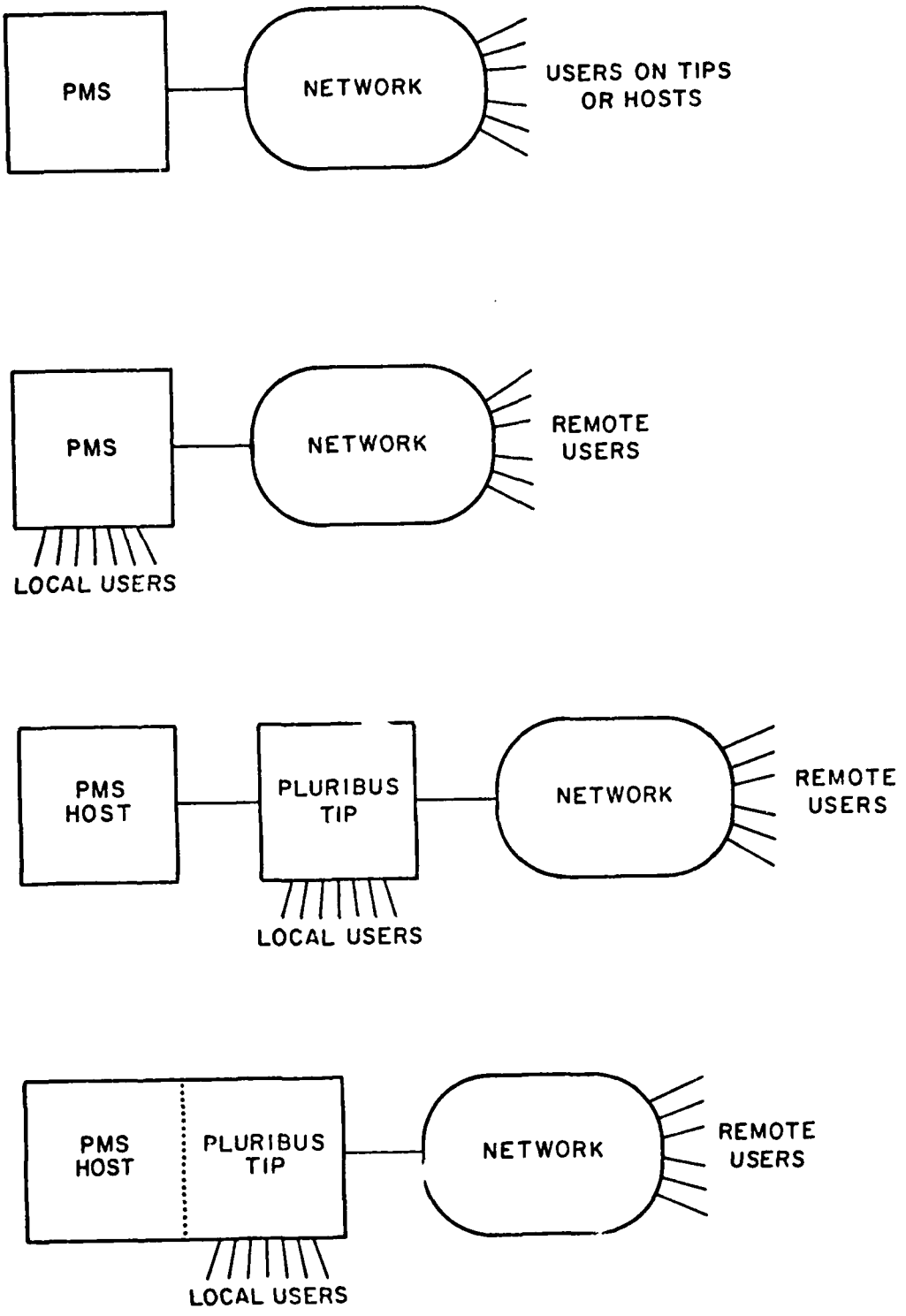


Figure 2 Terminal Access

system simple and permits the users of the message system to be dispersed geographically. The disadvantages of this approach are that: 1) it does not provide access ports for local users who would have to access the message system through the network; 2) it is somewhat inefficient in the way that it handles local users since they must make use of a perhaps unnecessary layer of network protocols; and 3) there are significant problems (discussed in Section 5) if the PMS is to be secure but the network is not.

The second approach is to support local terminals directly on the PMS. In the figure the PMS is also connected to a network, although it is quite possible that in some applications all of the users would be local and the network connection would be omitted. The disadvantage of this approach is that combining the message system functions and the terminal support functions in one logical and physical machine significantly increases the complexity of the system.

The third approach is to configure the PMS as a host without terminals as in the first solution described above, but to support the local users on a front end which is physically located close to the PMS host. A natural front end for this

purpose is a Pluribus TIP[3] which can support many local users and has the advantages of flexibility and reliability provided by the Pluribus architecture. The network connection in this figure is once again optional.

The fourth solution is to integrate the PMS host and the Pluribus TIP into one physical machine while retaining a logical separation between them. This is a reasonable extension to the previous approach, since it permits somewhat more efficient utilization of the resources (such as the Pluribus hardware) and it eliminates the external communications link between the two machines. While the physical configuration would be integrated, it is useful to think of the functions as segregated as they were in the previous configuration and to restrict the communications between the two portions of the machine to a nearly standard host interface. This approach has worked well with the IMP and TIP code in the system described in [3].

Of these four solutions, the first and the fourth are most attractive. The separate message system which supports no terminals but provides a service to a network is the simplest to

---

[3] W. F. Mann, S. M. Ornstein, M. F. Kraley, "A network-oriented multiprocessor front-end handling many hosts and hundreds of terminals", AFIPS Conference Proceedings 45, June 1976, pp. 533-540.

implement and is quite attractive in its own right. The fourth solution, which adds a Pluribus TIP as a front end to the message system, all combined in one machine, is also quite attractive, particularly in a secure environment.

For convenience, in the rest of this report we consider the PMS as a host which does not support terminals directly. When we refer to local terminals, we assume that they are connected to a Pluribus TIP front end which is physically integrated with the PMS but which is logically independent. We make this assumption for two reasons: First, it is a convenient way to build the PMS, a way which we recommend. Second, most of the design issues addressed in this report are independent of the choice made for terminal access method, since we recommend keeping the message system logically independent of the terminal front end, even if they are in the same physical machine.

2.2 User Programming

Current mail systems within the ARPA community are implemented as subsystems (and perhaps occasionally as pieces of the executive) operating on time-shared computers. This approach provides flexibility in that several message systems are available to the users of the facility; indeed, a user can even

write his own version of a message switching system if he chooses. The next overall design question is therefore: Should or should not a PMS host be user programmable? Presumably if the PMS were user programmable, methods for establishing user processes would be necessary, as well as protection mechanisms in the system to prevent undesirable interactions between user processes. On the other hand, if the PMS is not user programmable, these extremely complex mechanisms are unnecessary.

We believe that the PMS should not be user programmable. This decision simplifies the implementation, permits greater computational efficiency, and eliminates many of the serious security problems encountered in general purpose computer operating systems. This design decision is reflected in the rest of the system description in this report. If further user control of the user interface is a requirement for a particular application, then we recommend an interpretive approach such as that used in Hermes[4].

---

[4] T.H. Myer, C.D. Mooers, "Hermes Users Guide", BBN Report, June 3, 1976.

2.3 Network Interface

Some environments in which a PMS might be used require no network interface because all of the users are local, while in others a connection to a network is undesirable for reasons of security. Yet in many applications, connection to a communications network expands dramatically the usefulness of the PMS by increasing the number of users who can communicate through the system. Because of this flexibility, the message system design we recommend in this report includes a connection to a packet-switching network such as the ARPANET. Such a connection raises several questions. One is what type of host-to-host protocol the PMS should support. Another is how to achieve secure communication across the network to other secure installations, as well as how to achieve safe communications with unclassified users and hosts on the network.

Taking the ARPANET as an example, the host-host protocol on that network is probably not well suited to message switching functions. The NCP used in the ARPANET is oriented towards establishing and maintaining connections between communicating processes in hosts on the network. For the PMS application, connections are unnecessary and result in excessive inefficiency in the host as well as the network. A special protocol designed

for message switching such as[5] may be desirable. On the other hand, the standard ARPA protocols[6] are in widespread use and it seems essential that they be supported on the PMS if convenient communication from the PMS to the many existing ARPANET hosts is to be permitted. Fortunately, the network interface (ignoring for the moment its security impact) is one of the more modular pieces of the messages system. Thus, while it seems proper to us to assume the ARPANET protocols, it would be possible to substitute another (or to run parallel) if that later seemed better. We have already mentioned the suitability of [5] as one alternative. X.25[7], TCP[8], and the AUTODIN II protocols are other possible alternatives.

The second issue in the network interface is related to secure operation across the network. A PMS could presumably be connected (with proper safeguards) to the unsecure network, provided that all of the messages going to or from users on the

---

[5] "MSG: The Interprocess Communication Facility for the National Software Works," BBN Report No. 3237, January 1976.
[6] "ARPA Network Current Network Protocols," Stanford Research Institute, December 1, 1974.
[7] A. Rybczynski, B. Wessler, R. Despres, and J. Wedlake, "A new communication protocol for accessing data networks--The international packet-mode interface," AFIPS Conference Proceedings 45, June 1976, pp. 477-482.
[8] V.G. Cerf, R.E. Kahn, "A protocol for packet network intercommunication", IEEE Transactions on Communications, Vol. COM-22 5, May 1974, pp. 637-648.

network were unclassified. This in itself may be quite valuable; for example, if such a PMS were installed on the ARPANET, all of the secure users could have dedicated and encrypted lines directly to the PMS, and unclassified users could access the PMS through the network or through local connections to the PMS. By doing this, the secure users would have one point of contact with their network mail for both secure and unclassified communications. Clearly this requires communications security (KGs, etc.) for the secure access lines and multi-level security to permit both unclassified and classified users to use the PMS, both of which the PMS will accommodate.

An extension of this concept to permit secure communications across the network would require some form of encryption for secure traffic. This could be provided by installing a secure PLI[9] between the PMS and the network. A natural implementation in this case would be several PMS sites throughout the country. At each site most of the communications would be between the local users at that site; however, occasionally there would be communications between the sites. If this is combined with an unclassified port to the network, then the users on any PMS would

[9] "Specifications for the Interconnection of a Host and an IMP," BBN Report No. 1822, January 1976, Appendix H.

be able to communicate with any classified or unclassified user on the network. As another option, secure terminal concentrators (perhaps secure PTIPs)[10] could be developed; such a device could access the PMS across the network. This is clearly the most flexible and powerful approach and is the one that we recommend as the long-term configuration. However, the availability of a suitable secure PLI is necessary before this can be implemented.

These issues are discussed further in Section 5.

2.4 Mass Storage

The Pluribus currently supports only core memory and one type of small disk. Although the memory address space accommodates a half million 16-bit words of memory, for many applications, including that of the PMS, this is not adequate. For the purposes of the PMS, it is clear that effective mass storage is required. We have investigated two ways this capability can be obtained: disks and electronic beam addressable memories (EBAMs).

---

[10] Mann et al, op cit.

The disk which is currently interfaced to the Lockheed SUE (the processor used in the Pluribus) is unsuitable for the PMS because of its small size (two megabytes) and its inadequate interface, which is not well matched to the Pluribus since it uses conventional interrupts and 16-bit memory addresses, both of which are incompatible with the Pluribus environment.

Fortunately, the peripherals industry now seems to have reached a consensus on a range of disk sizes. Several manufacturers are supporting a line of disk drives (similar to the IBM 3330) with compatible interfaces. These disks range in size from 40 megabytes to 300 megabytes. In many cases the drives themselves are upward compatible and can be field changed to higher capacities as required. By virtue of the popularity of these devices and the vendors' commitment to this wide range of machines, it seems likely that they will be around for some time and that as new advances in technology occur, they will be included in compatible ways. Examples of these disk drives (besides the IBM 3330) are Calcomp Trident, Control Data Storage Modules, Ampex 900 and 9000 series, Diablo 410 series, and Memorex 670 series.

Several commercially available universal controllers have also evolved, such as the Telefile DC16C, which (by changing a

single card) can interface to any of several disk drives while presenting the same interface to the computer. The controller takes care of such issues as disk addressing, formatting, address verification, defective track relocation, error correction and detection, data rate buffering, and diagnostics. It remains only to develop and construct an interface from this controller to the Pluribus. Such an interface would map the various status bits and control words into a standard Pluribus 8-word device control block[11]. It would contain the necessary logic for full 20-bit system addresses so that data to and from the disk could be written to any area in common memory. The interface would also contain provisions for connection to more than one I/O bus so that even in case of bus failure, the disk would still be available.

The mass storage requirement for a PMS is on the order of 150 million characters. This value is based on an assumption of 15,000 messages per day from 2000 users with 1000 characters per message and disk storage for 10 days. Details that lead to these numbers are presented in Section 7. The characteristics of a disk which would support these requirements are:

---

[11] Pluribus Document 2, "System Handbook," BBN Report 2930, p. 37 ff.

- capacity - 300 million characters
- peak transfer rate - 1.2 megabytes per second
- access time - 10-55 milliseconds, 30 average
- rotation time - 16.66 milliseconds
- cost - .01 to .02 cents per character

Although such a disk is a cost effective means of mass storage, it requires a relatively long time to access records. This is a serious problem should it be necessary to use such a device for temporary data storage or swapping. A promising technology for mass memory which has recently emerged is EBAM memory, in which the bits are stored in a MOS structure that is addressed by an electron beam. This approach has the potential for very high capacity with costs similar to those of a small disk but with access times comparable to core memories. There are at least two vendors: Microbit in Lexington, Mass.[12] and General Electric[13]

The Microbit memory module has 4 million 16-bit words (actually there are 6 extra bits per word for error correction).

---

[12] D.E. Speliots, "Bridging the Memory Access Gap", AFIPS Conference Proceedings 44, May 1975, pp. 501-508.
[13] General Electric, "BEAMOS (BEam-Addressed Metal Oxide Semiconduction)," undated G.E. internal non-proprietary memorandum.

The controller can handle up to 8 such modules. Within each module, data is stored in blocks with 1024 words to a block. It takes 20 microseconds to access a block, and the peak data rate is then about 4 megabytes per second. Writing is four times slower. In addition, there is a 10% overhead for refreshing since the memory gradually decays while used if not occasionally refreshed. When the memory is not being used, no refresh is needed. Even with the power off, data will be valid for several weeks. The overall error rate after all internal corrections is on the order of 10**-11 to 10**-12[14]

The manufacturer-provided controllers for these devices would have to be interfaced to the Pluribus. From the Pluribus side, this interface would look just like a fast disk, except that seeks are 3 orders of magnitude faster. The controller performs all addressing, refreshing, and error correction functions, and also provides a buffer for data rate matching.

Mass storage is required in the PMS application primarily for two purposes: swapping working storage, and short and medium term file storage. There may also be some ancillary functions such as program storage. In a small PMS, economics probably

[14] In this report we use a double asterisk to denote exponentiation.

dictates that we use a disk to provide all of these functions. As the systems become larger and the demands on the swapping channel become more intense, it may make sense to substitute for the disk a EBAM memory with its much lower seek times for swapping from a disk. As the system grows even larger, the relatively limited capacity of the EBAM memory may prove to be a hindrance in the storage of large files, and we would then recommend a system which has both EBAM memory and disks. The disks would be used primarily for file storage where the longer access times are acceptable, and the EBAM memory could then support swapping at very high rates.

We recommend that the prototype PMS be based on swapping from a disk, to eliminate the uncertainties associated with the still developing EBAM technology. At the same time, progress in this technology should be monitored closely so that a proper decision can be made about using it when the time comes to build large systems.

3. THE SOFTWARE FOR THE PLURIBUS MESSAGE SWITCH

The software in the PMS defines the interface seen by the system's users, provides medium- and long-term storage of messages for its users, transmits messages between users both at the local site and at other sites, and provides various other services. This section discusses those aspects of the PMS software that present new challanges, ignoring areas that are well understood generally. We deliberately omit discussion of one very important part of the design of any message system: the details of the interface that it presents to its users, i.e., does it look like MSG, Hermes, RD, or some other message system. There are two reasons for this omission:

(1) There are ARPA-sponsored studies currently underway both at BBN and elsewhere to determine the nature of this interface, studies which we did not attempt to duplicate.

(2) We have had no difficulty proceeding with this design study without knowing the details of the user interface. (Incidentally, if we had to choose one existing message system to recommend for implementation on the Pluribus, we would choose MSG because of its reasonably small size and because it is reasonably powerful.)

What we have done is to make some assumptions about the overall properties of the user interface, assumptions which we know to be valid for all existing message systems that we are familiar with. (These include Hermes[15], MSG[16], and SNDMSG/Readmail[17].) These are the properties which we have assumed:

- There is a facility for constructing messages. It can be used to prepare the various header fields ("To:", "From:", "Cc:", etc.) and to compose and edit the body of the message.

- A message once composed may then be transmitted, both to other users and to the files of the originator.

- Received messages may be filed, with a structuring imposed by the user.

- There is a facility for forwarding and answering a received message to another person, perhaps accompanied by annotations.

---

[15] T. H. Myer, C. D. Mooers, op. cit.
[16] J. Vittal, USC Information Sciences Institute, MSG on-line documentation.
[17] "TENEX User's Guide," Jerry D. Burchfiel, et al, Bolt Beranek and Newman Inc., Cambridge, MA, January 1975 revision, p. 137 and pp. 165-170.

- There is a facility for examining files of messages. It provides both for reading newly received messages and for looking at previously filed messages. It can process requests like, "Show me all messages from Jones about UFOs." (Presumably such a request would be expressed in a less English-like form.)

- There will be various housekeeping facilities. For example, there may be an archival store, a place to store older material for which economy of storage cost is more important than immediate access, and commands are needed to control this feature.

One other point is relevant. Frequently for expository convenience or to aid our thinking it has been useful to consider a specific message system. In all such cases we have considered Hermes, checking carefully to insure that we were not thereby precluding any other system. We have so used Hermes both because it is convenient, being an in-house development with all the needed expertise nearby, and also because it is possibly the most sophisticated of the currently proposed systems. Since our design will support Hermes, it will surely support simpler systems.

The remainder of Section 3 presents, in turn, the file system that will support the PMS, the swapping system, protocols and formats, and certain housekeeping issues of interest. As mentioned above, these are areas in which we think that we have something novel to contribute.

3.1 The File System

The file system in the PMS performs the same task as does the file system in a time-sharing system, i.e., it provides for medium term storage of named data in a manner that insulates the user from the problems of hardware addressing. Whereas a conventional file system provides for storage of arbitrary binary data whose structure is of no concern to the file system, the special requirements of the PMS make it appropriate to construct a file system especially for the purpose. By tailoring the file system to the job, we expect both a more efficient product and significant economy in the implementation, in each case because we would be building a special purpose tool rather than one with unneeded generality. Thus the file system knows about messages and includes special facilities for dealing with them. We feel that the limited and specialized requirements justify designing a file system tailored to the PMS application.

The basic organization is conventional. Associated with each user is a directory (DIR) holding all information possessed by the PMS about that user. The DIR provides a mapping of names chosen by the user onto data stored in the file system. The names must be unique for a given user, although there is no problem if several users use the same name for an object.

The principal purpose of the file system is to store messages, including those that the user has received as well as file copies of messages that he has sent. It is not adequate just to store all of these messages -- the user must be able to impose his own structuring on them. This structure is provided by association lists (ALs), each of which is a named entity (an entry in a DIR like a file in a time-sharing system) which is an ordered collection of messages. For example, a user might have an AL named "FROM-JONES" to hold all messages from Jones. Presumably the messages would be in chronological order, although the user has complete freedom in this matter. A user can have many ALs, and a given message might appear in several of them (although as explained below only one copy of each message actually exists in the file system). Every DIR holds an AL named INBOX, the repository into which incoming messages are placed, like an IN-basket on one's desk. Reading new messages is merely the process of examining the contents of the AL named INBOX.

The file system also deals with owned objects (OOs). An OO is any piece of data (other than a complete message) which a user wishes to store. It might be a piece of text he plans to include in several messages, it might be a pre-stored address list for a "To:" field, etc. Alternatively, it might be information associated with control of the PMS, such as the filters and templates of Hermes. Some OOs, for example the address list for a project, may be shared among several users. Associated with each OO, although invisible to the user, is a list of all users entitled to access it.

Finally, associated with each user and stored in his DIR are certain user data. Section 3.1.4 contains further details.

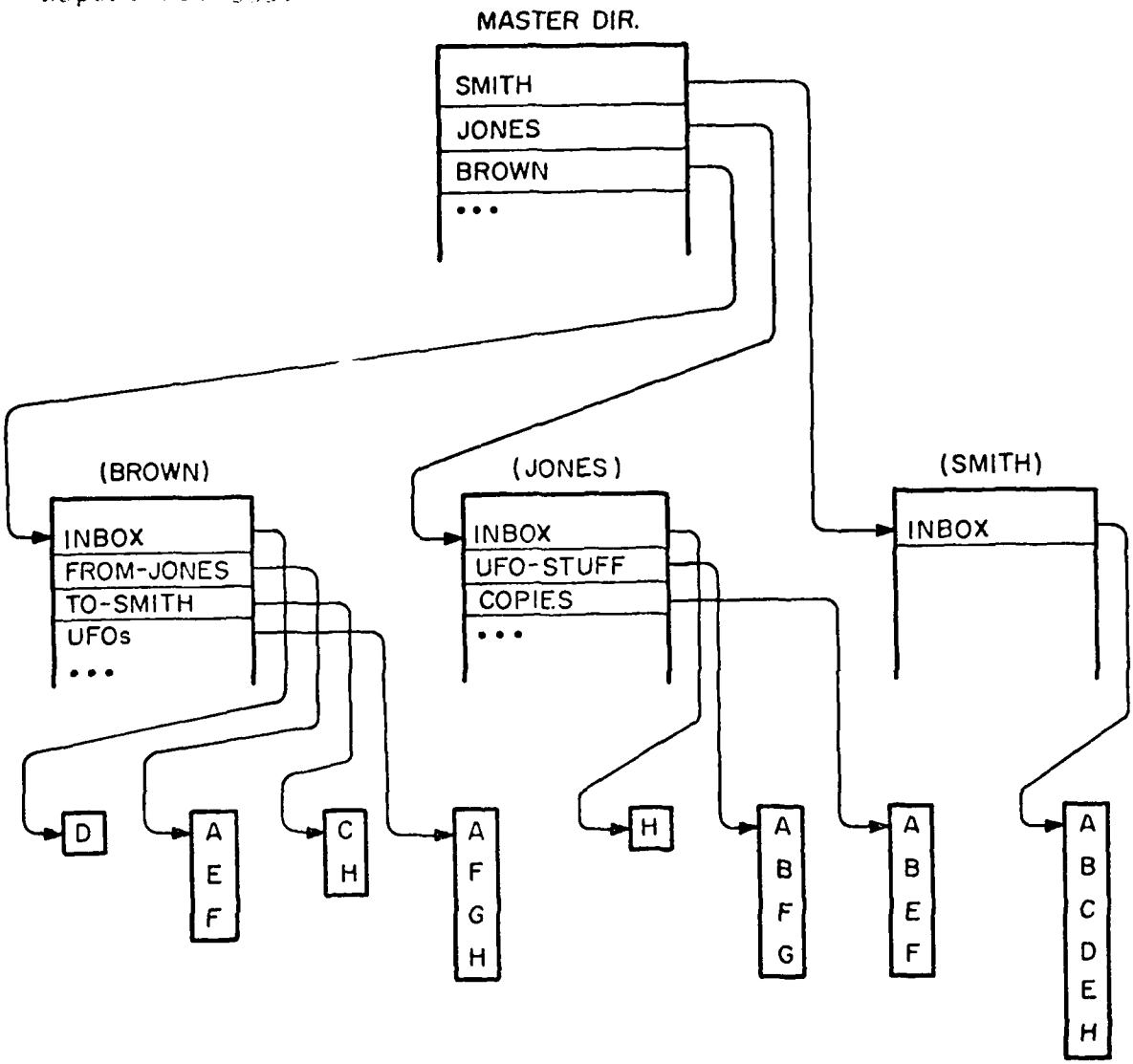
There are two key aspects of the storage of messages in the file system:

- There is only one copy of each message at each site, no matter how many ALs the message appears on. This is true even if the ALs belong to different users.

- A message once created is never changed. This is almost implied by the first point, since all "owners" of the message share the same copy.

Implications of these points permeate much of the following discussion. Consider a message composed and sent to three users all at the same site as the sender, with a file copy retained by the originator. The process of sending the message after it is composed involves merely placing pointers to it in four ALs (three INBOXes and the AL for the file copy). If an addressee is located at another site, then a copy of the message must be sent to that site. However, only one copy need be sent to that site even if there are multiple addresses there.

The directory structure is suggested by Figure 3. This shows a master directory with entries for three users, Smith, Jones and Brown. Parts of each of these users' DIRs are also shown. Each has an AL named INBOX, as well as some other ALs as shown. (No OOs are shown in this diagram.) Each AL is shown as an ordered list of pointers, letters being used to represent pointers to the labelled messages shown at the bottom of the figure. Note message A, to Brown from Jones with a copy to Smith, on the subject of UFOs. The originator Jones has saved a copy in the ALs UFO-STUFF and COPIES, the latter holding copies of all messages he has sent. Brown has filed the message under FROM-JONES and under UFOs. Smith merely lets all of his messages collect in his INBOX.



A: TO: BROWN
FROM: JONES
Cc: SMITH
SUB: UFOs

B: TO: SMITH
FROM: JONES
SUB: UFOs

C: TO: SMITH
FROM: BROWN
SUB: X

D: TO: BROWN
FROM: SMITH
SUB: X

E: TO: BROWN
FROM: JONES
Cc: SMITH
SUB: Y

F: TO: BROWN
FROM: JONES
SUB: UFOs

G: TO: JONES
FROM: BROWN
SUB: UFOs

H: TO: JONES
FROM: BROWN
Cc: SMITH
SUB: UFOs

Figure 3 Directory Structure

Although the provision of security in the PMS is discussed in detail in Section 5 of this report, certain points are best made here. Each message and each OO is stored with a checksum[18]. As mentioned previously, each OO has associated with it an access list indicating which users may access it, and this access list field is also part of the checksummed data. Two points are important: First, each object stored in the file system includes as part of it a list of all users who are entitled to see that object. For a message it is all users named in the address fields, and for an OO it is those listed in the access list. The second key point is the checksum, the correctness of which guarantees the integrity of all the bits of the object, including in particular those bits that indicate who may have access to it. These points provide the requisite handle to permit the security system to guarantee that no data is given out except to a user entitled to it.

3.1.1 Association Lists (ALs)

The user thinks of an AL as an ordered set of messages, although it is implemented as an ordered set of pointers to

[18] The implementation we propose in order to achieve security requires two checksums on each message. This fact is ignored in the rest of this section.

messages. An AL belongs to only one user, and it has a name which is an entry in the user's DIR.

An AL named INBOX appears in every user's DIR. This is his "IN-basket", and messages addressed to him are deposited there. The user may create other ALs and name them as he chooses. For example, if he has one AL for all messages from Jones and another for all messages about the UFO project, he might file a message from Jones about UFOs in both ALs. (Of course, there is only one copy of the message and it is pointers to it that are placed in the ALs.) After doing so, he might then delete the message from INBOX. (That is, the pointer to the message is deleted from that AL.)

3.1.2 Owned Objects (OOs)

An OO is a named object belonging to a user, its name appearing as an entry in the user's DIR. An OO can hold any piece of data, other than a complete message which is afforded special treatment, that a user may want to store in the file system. Possible kinds of information that might appear in an OO include the following:

- pre-stored data for an address field. This might be a long address list for a project. It could be included in

the "To:" field of a message being composed with a facility similar to the CTL-B feature of Hermes or SNDMSG.

- a long piece of text to be included in the body of several messages.
- save status while editing.
- housekeeping items appropriate to the user interface, such as the templates and filters of Hermes.

Each OO has a name which must be distinct from the name of any other OO or any AL in the user's DIR. The entry in the DIR for that name contains a pointer to the place where the OO is stored in the file system. For redundancy as well as for the benefit of the security system, each OO has stored with it (although invisible to the user) the names of those entitled to access it.

3.1.3 The Message

A message is a complete communication from a user to one or more recipients. It is a self-contained object, in which the originator and all addressees are given explicitly. Thus it is possible by examination of a message to determine with confidence which users are entitled to access it.

Each message is pointed to by one or more entries in ALs. The disk address for a message that appears in an AL is the address of a record such as the one shown in Figure 4. What is pointed to is a record holding the various header fields and pointers to the records holding the text. The first few words hold a flag indicating that this record is a message header, a checksum, and other data. The next words are pointers, one for each header field. The datum is the offset from the beginning of the record where the field is stored. All header fields are stored in the header record itself. (There will be provision for multiple header records where necessary.) For the text part of the message, though, the header record contains not the text itself but disk addresses where the text is stored. The flag "1" indicates such a disk address, and "0" indicates the end of the list of addresses. This method of storing messages makes context searches on header fields particularly easy, since there is ready access to each field without text scanning. We see in section 3.3 that this format matches well the format used for transmission of messages over a network.

A forwarded message requires special treatment. Message forwarding involves sending a copy of a message to another user, perhaps accompanied by comments or other annotations. In many

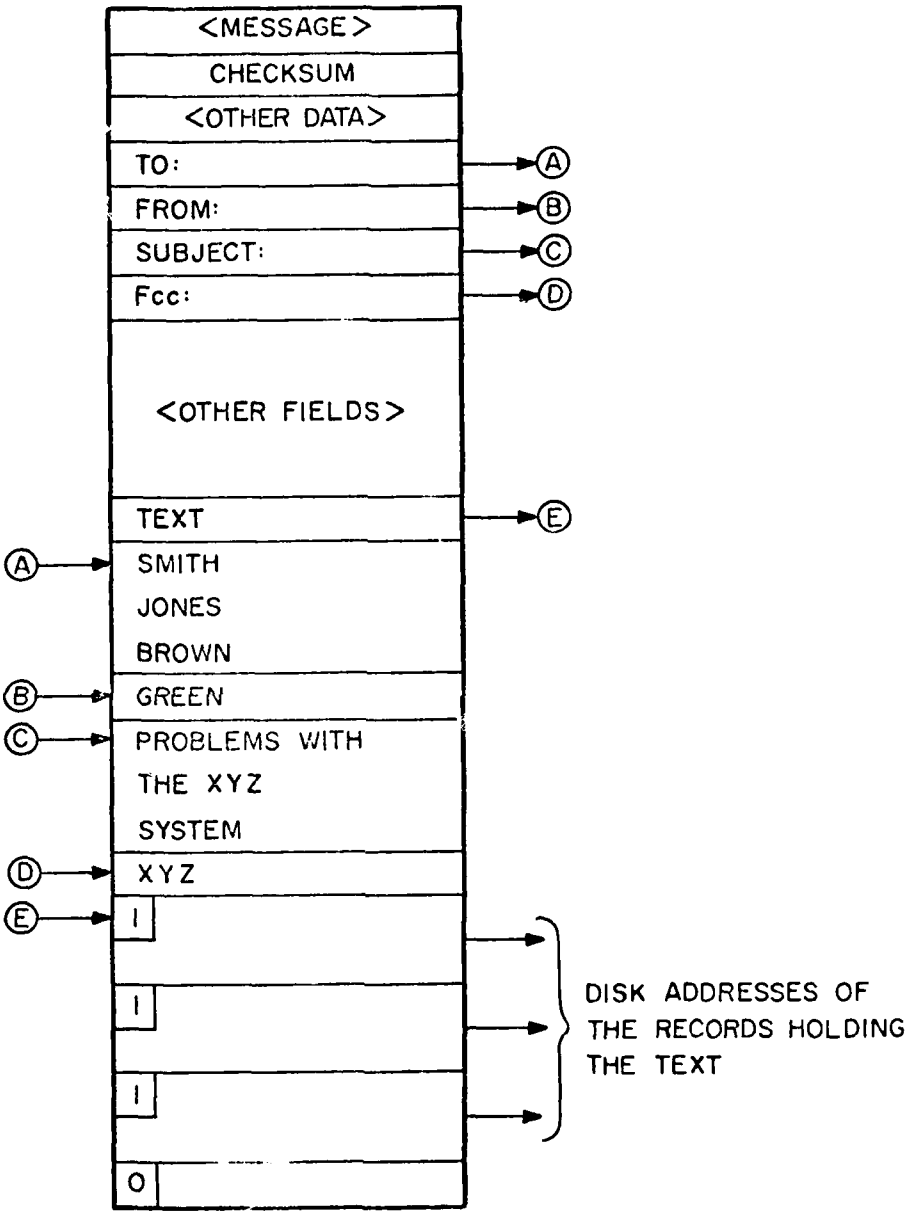


Figure 4 Format of a Message in the File System

applications the usual mode of operation involves frequent forwarding of messages. For example, in the military it is often standard operating procedure that all messages to a facility be addressed to the commanding officer. Each received message is then forwarded down through the chain of command to the action recipient, frequently with copies to various information recipients. Since this is anticipated to be a common occurrence in the use of the PMS, it is important that our implementation handle it efficiently.

The simplest way conceptually to handle a forwarded message is to include a complete copy of it in the text field of the new message. Thus Smith, having received a message which he wants to forward to Jones, composes a new message addressed to Jones which includes in its body all the headers and text of the original message. The obvious problem, particularly in an environment in which frequent forwarding is the way of life, is the space required in the file system for multiple copies of a message. It therefore seems appropriate to provide for forwarding as a special case. Rather than include all of the original message in the forwarded message, we merely include a pointer to the original. Note now Figure 5, which shows the structure of a forwarded message. This differs from the structure of a simple

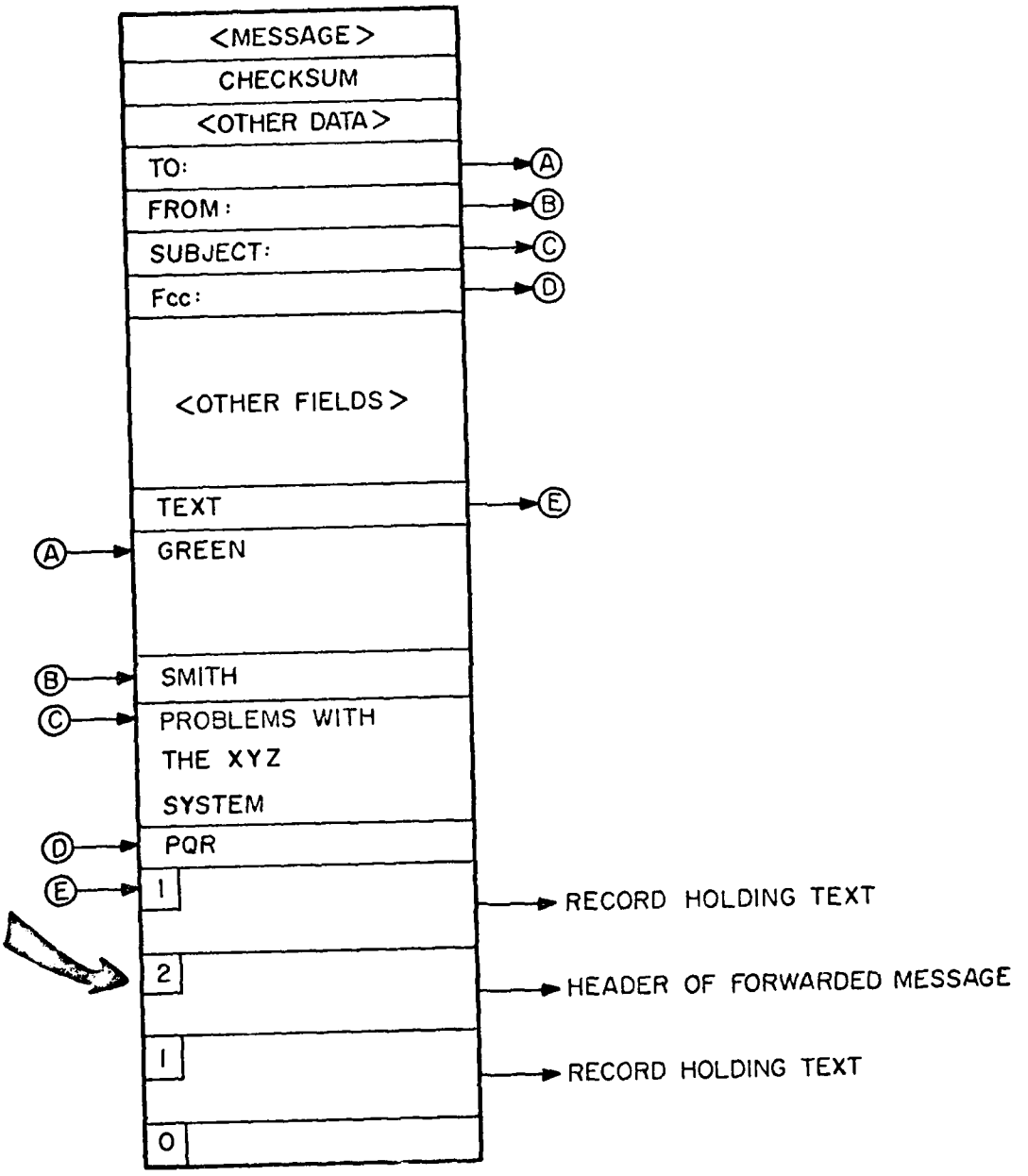


Figure 5 A Forwarded Message

message as shown in Figure 4 only in that one of the text pointers points to a message rather than to a record holding text. In particular, note the second disk address in the text field (marked with a pointer to it). This has a flag of "2" rather than "1", indicating that it is a pointer to an entire message rather than to a text block. This implementation makes annotation of a forwarded message convenient, since the annotation can be a record of text that precedes the forwarded message. In the forwarded message in the figure, there is apparently annotation both before and after the forwarded message.

This implementation has an implication for security. Recall Smith, who wishes to forward a message to Jones. The original message was not addressed to Jones and so does not show him as an addressee. Our earlier discussion (see page 37) suggests a scheme that releases an object to a user only if he is entitled to it. This mechanism must therefore be cognizant of forwarded messages, so that releasing the message from Smith to Jones includes releasing the forwarded message it points to. There is no particular difficulty with this, but it must be built in as a special case.

There is an implementation problem with any pointer structure such as the one just described: the possibility of circular list structures. In the present scheme, it is inherently true that such circular structures cannot be created, since a message can point only to an already existing message and no message once created can be altered. Regardless of the unquestioned truth of this claim, for adequate robustness it is necessary that the code be prepared to deal with a circular structure should one come into existence by some means. This point is addressed further in Section 3.4.

3.1.4 User Data

Certain data are stored in the file system for each user as system data, in addition to the user data such as ALs, messages and OOs. The system data may be stored either directly in the DIR or in a record pointed to by a word in the DIR, depending on how much of it there is. Such data items include the following:

- classification level. In the simplest case this is the highest classification level the user is cleared to. If a need-to-know scheme is implemented, the relevant data are stored here.

- name association data. The user may specify how he wants to refer to certain individuals. For example, he may want to say just whom he means when he addresses a message to Smith. Such data are stored here.

- sending restrictions. For security or other administrative reasons, it may be desirable to restrict the set of users to whom a given user may send messages.

3.1.5 The Backup System

It is of great importance in an application such as the PMS to minimize the impact of any system failure, either hardware or software. The file system is implemented as storage of data on some physical device, and that device might fail catastrophically (for example, by physical destruction of a recording surface by a recording head). The backup system is provided to minimize the impact of such a catastrophe. This is in addition to the Pluribus philosophy of robustness that ensures quick recovery from small failures. Our plans for backup are discussed in Section 4.2.

3.2 The Swapping System

It is unlikely that the main memory available in the PMS will be adequately large to hold simultaneously all of the program as well as all data needed for all active users. It is therefore necessary that some needed information be stored in a larger but slower memory device while it is not in active use, being fetched to main memory when needed. This device may be the same disk used for the file system, but in a many-user application the high bandwidth required for swapping may necessitate a faster device.

While it is possible to swap either program or data or both, in the PMS we intend to keep all of the program in memory at all times and to swap only data. This approach provides significant conceptual simplifications in the system design, since the program knows when it is about to access new data and can make explicit arrangements to get it into core. Use of the alternate approach of permitting parts of the program to be out of core gets one involved in the complexities and costs associated with demand paging.

We will perform swapping in the following way: When a program reaches a point where it needs user data, it first checks

to see if the data are in core. If not, it places on a queue a request that the data be fetched and terminates its execution, first insuring that it will run again when the data are available. Further, when a program is done with a data block, it so indicates by putting a write-out request on a queue. Of course, if a program needs data that are still in core waiting for a write-out request to be processed, then that write-out request is canceled. This scheme is simple and workable and should be easy to program and debug, in contrast to demand paging which is quite complex. In return, of course, we put the burden on the application programmer, a burden that demand paging frees him from. This seems to us to be the proper decision in a large single-application program such as this one.

3.3 Protocols and Formats

There are two different kinds of communication over the network, using different protocols. One is complete messages moving over the network from one PMS site to another; the second is character data representing user I/O to and from the PMS. Both will be transmitted using packet switching technology as developed for the ARPANET.

In a dedicated message system such as this one, it seems desirable that the format used for transmission of messages over the network be similar to that used for storing a message in the file system, both for conceptual clarity and so as to minimize the processing required to send or receive a message. The format used in the present ARPANET is pure text, a format that has the disadvantage that locating a particular field requires textual scanning. Figure 6 shows the format which we intend to use for a message transmitted between sites. It is quite similar to the format of a message in the file system, as shown in Figure 4. However, the pointer indicated by E points to the text itself, which is the rest of the message. Note that the length of a transmitted message is arbitrary and is not restricted to the length of a disk record. (It is a simple task to convert messages into and out of the conventional ARPANET textual format when it is necessary to communicate with existing mail systems. Of course, implementing the FTP protocol required for conventional message delivery is less easy.)

If the message being transmitted contains a pointer to a forwarded message, then the latter must be sent too so that the receiving site can make all of the pointers correct. If a copy of the forwarded message already happens to exist at the

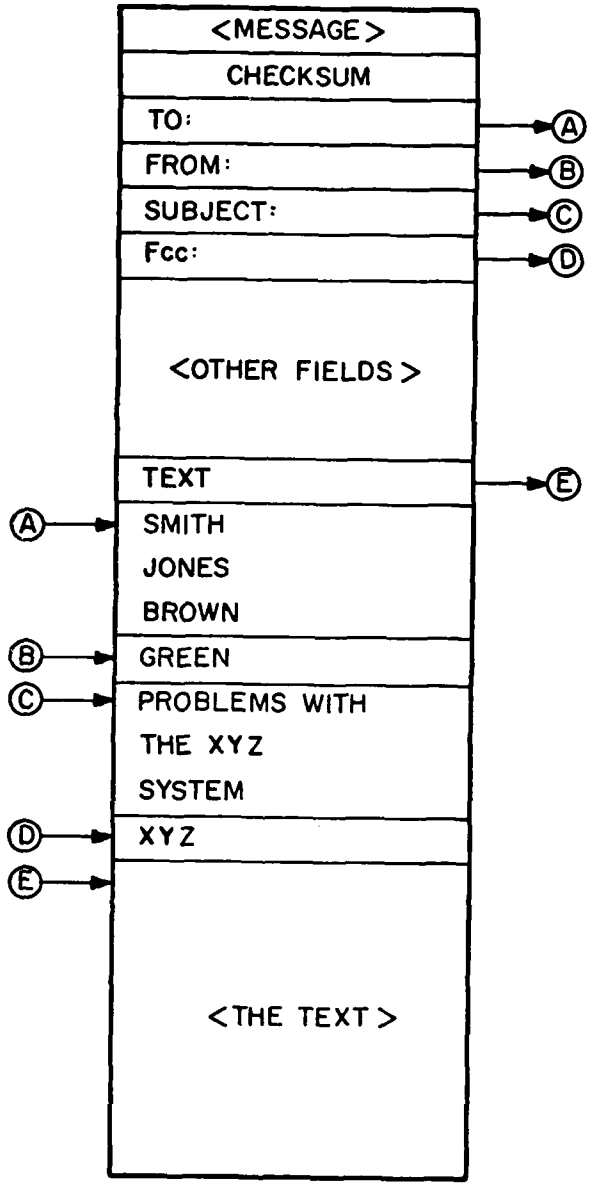


Figure 6 Message Format for Sending

receiving site, there is the possibility that two copies will then exist. Although this can be avoided with care, doing so is probably not worth the extra trouble.

The other protocol of interest is for transmission of terminal I/O. Although the Telnet protocol used on the ARPANET might be used, this is inefficient in use of network bandwidth and processing bandwidth in certain applications. Alternatively, one might use a protocol similar to that used by the PTIP for communication with TENEX. The problem with the Telnet protocol is that only a few characters of data are accompanied by many bits of address and header information. Without going into details that are not relevant here, the PTIP collects in a buffer characters for many users on each host and transmits all of them together, either when the buffer becomes full or periodically. Thus, the addressing and header overhead cost is distributed among many characters, as is the processing bandwidth, rather than being imposed on only a few characters. The Telnet approach is an appropriate one if the originating site is sending characters to a few users on each of many other hosts, while the PTIP protocol comes into its own if the originating site is sending characters to many users on only a few hosts. We suspect the situation in the PMS application will probably involve a

comparatively small number of sites, and thus the PTIP protocol should be used. In any case, the Telnet protocol will have to be supported for communication with the large numbers of terminals which already exist in the ARPANET, for instance.

3.4 Housekeeping Issues

Two non-trivial problems in the PMS are deletion and archiving of messages. A user may delete a message from an AL, or he may indicate that a given message need not be kept in the file system but may be moved to the archival store. Because messages are shared, the processing of these requests requires care.

In any system such as the file system in the PMS that involves objects and pointers, it is necessary to retain any object that is pointed to. In other words, any message that is pointed to by at least one AL or a forwarded message pointer must not be deleted from the system. Thus a deletion request must not cause the actual removal of a message until the message has been deleted from all places that once pointed to it. There are various ways to implement this, one of the simpler being a reference count scheme. Fortunately, this simple scheme is adequate for the PMS application. It works as follows. Part of

each message (stored in a field in the message header) is a reference count which holds the number of pointers to the message that exist. The number is incremented by one each time the message is added to a new AL or each time the message is forwarded. The number is decremented by one each time the message is deleted from an AL or a forwarded message containing a pointer to it is deleted. When the number reaches zero, it is safe to delete the message. Care must be taken in coding the algorithm so that it is fail safe, in the sense that a system stoppage during its execution will not leave the data in a dangerous state. For example, the pointers to a message must be removed before the reference count is decremented.

The usual hazard with a reference count scheme is circular lists. However, the nature of the application is such that circularities are not possible. A message never points to an AL, so the only possibile problem is a circle of forwarded messages. But this cannot happen, since a forwarded message can point only to an already existing message, and no message once created can be changed. Even if a bug permits a circle to be created, the only harm will be that a message cannot be deleted and the space it occupies is lost temporarily.

Archiving is a related problem. A user may indicate that a given message may be moved to the archival store, but actual moving can take place only after all users concerned with that message no longer want it. Thus an archive pending flag must also be maintained for each message, the flag indicating that at least one archive request is pending. The transfer to archive storage takes place when the reference count reaches zero and the archive pending flag is on. Again, care is necessary. For instance, consider a message appearing on two ALs. User 1 requests that it be archived, and then user 2 says to delete it. We must be sure that the deletion request causes archiving and not deletion.

It is possible that messages and other records on the disk can get lost, so that they appear in no part of the file system and are not on the free list. Although this is a benign sort of failure that does not hurt performance much or compromise security, it is important that such lost space be eventually recovered. This is consistent with the usual Pluribus philosophy of checking for things that "can't possibly happen", and fixing them if they do. The method is conceptually simple but organizationally complex. The idea is to find any disk record that is not pointed to by anything and is not on the list of free

records. First one makes a pass through the disk, setting a flag bit in every record to OFF. Then one looks at every DIR, at every AL and every OO in each DIR, at every message in each AL, and at every forwarded message pointed to. Further, every disk record on the free list is looked at. Every disk record looked at in this process has the flag turned ON. Finally, a complete pass is made through every disk record. Each one with the flag OFF has become lost somehow, and it is placed back on the free list.

Various improvements can be made on this scheme, but they are not worth going into here. For example, the reference counts should all be recomputed during the process. The hard problem is that the system does not hold still while this process is going on but is processing traffic the whole time. Thus considerable complexity is added to the problem. The needed algorithm is similar conceptually to that part of IMP reliability code that finds lost buffers, an algorithm which we developed and understand.

Note that our approach here is consistent with the usual approach taken in Pluribus software. We design the system as best we can so that it will not fail, and then we check for failures anyway and fix their bad effects. It is this philosophy

that enables us to produce systems that keep running in spite of random failures that cannot be predicted.

4. SYSTEM AVAILABILITY AND RELIABILITY

In this section we discuss the issues of message system availability and reliability. First, however, we make clear what we mean by these two terms. Availability is a measure of how much of the system is available to be used when users want to use it. Reliability is a measure of the accuracy of the system. For instance, if there is a power failure and there is no backup power or system, then the system is unavailable. If the system is available and it miscalculates the result of a user request or misfiles user data, then the system has made an error and is unreliable to that extent. Of course, there are some obvious couplings between reliability and availability. If the system makes too many errors (i.e., is too unreliable) from the user's point of view it might as well be unavailable. The important point to be made is that many techniques which improve reliability do not necessarily also improve availability and vice versa. Note that we define security to be a separate problem from reliability and availability; maintaining a high level of reliability or availability does not ensure security. Of course, unreliability may impact security, so the security techniques must guard against unreliability. (Security is discussed in Section 5.)

In the rest of this section we discuss, in turn, standard Pluribus techniques meant to insure availability, additional techniques to attempt to provide complete system reliability, distributed techniques to improve availability beyond what is possible with a single system, and techniques to insure user-level reliability.

4.1 Standard Pluribus System Availability Techniques[19]

Computer reliability is a common, serious, and difficult problem which has been approached in many ways. For critical applications (e.g., space exploration), large amounts of money are spent to overcome such apparently trivial weaknesses as problematical power supplies and connectors. Although a great deal of attention is given to tailoring computers to particular job environments, the commercial world of computer manufacturers has provided no adequate answer to the reliability problem.

The notions of fault-tolerant and fail-soft systems have been around for a number of years and because reliability is such

[19] Additional material on the topics discussed in this section can be found in S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Heart, "Pluribus -- A Reliable Multiprocessor," AFIPS Conference Proceedings 44, May 1975, pp. 551-559. Also, see Appendix B for a summary of the basic Pluribus structure.

a crucial issue in a communications network, it was decided that some of these ideas should be exploited in the design of the Pluribus.

The availability goal of the Pluribus is not that the system should never break, but rather that it should recuperate automatically within seconds or minutes from most troubles and that the probability of total failure should be dramatically reduced over traditional machines. The system should survive not only transient failures but also solid failures of any single component. It is assumed that it is not necessary to operate correctly all of the time so long as outages are infrequent, kept brief, and fixed without human intervention.

4.1.1 Appropriate Hardware

The Pluribus structure provides hardware integrity through the following principles. Not only are duplicate copies of a particular resource provided, but it is also necessary to assure that the copies are not dependent on any common resource. This means in the Pluribus that in addition to providing multiple memories, there are multiple busses on which the memories are distributed. Furthermore, each bus is not only logically independent, but also physically modular. The chassis, with its

own power supply and cooling, is built into an integral unit which may be powered down, disconnected, and removed from the rack for servicing or replacement while the rest of the machine continues to run.

All central system resources, such as the real time clock and the PID, are duplicated on at least two separate I/O busses. All connections between bus pairs are provided by separate bus couplers so that a coupler failure can disable at most the two busses it is connecting; all other interconnections between busses are unaffected.

When a particular communications circuit is deemed critical, it is connected to two identical interface units (on separate I/O busses), either of which may be selected for use by the program. When the extra reliability is not worth the extra cost, the line is only singly connected.

In order for the system to adapt to different hardware configurations, facilities have been provided which make it convenient for the software to search for and locate those resources which are present and to determine the type and parameters of those which are found.

To allow for active failures, all bus couplers have a program-controllable switch that inhibits transactions via that coupler. Thus, a "malicious" bus may be effectively "amputated" by turning off all couplers from that bus. These switches are protected from capricious use by requiring a password. Naturally, an amputated processor has no access to these switches.

4.1.2 Software Survival

With the above features, the Pluribus hardware can experience any single component failure and still present a runnable system. One must assume that as a consequence of a failure, the program may have been destroyed, the processors halted, and the hardware put in some hung state needing to be reset. Three broad strategies have guided the means used to restore the algorithm to operation after a failure: keep it simple, worry about redundancy, and use watchdog timers throughout.

4.1.2.1 Simplicity

First, all processors are identical and equal; they are viewed only as resources used to advance the algorithm. Each is able to do any system task; none is singled out (except

momentarily) for a particular function. A consequence of this is that the full power of the machine can be brought to bear on the part of the algorithm which is busiest at a given time. A further consequence is that should any processor fail, the rest will continue to perform the necessary tasks, albeit at reduced capacity.

A second system characteristic which arises from a desire to keep things simple is passivity. The terms active and passive describe communication between subsystems in which the receiver is expected to put aside what it is doing and respond. The quicker the required response, the more active the interaction. In general, the more passive the communication, the simpler the receiver can be, because it can wait until a convenient time to process the communication. Neither the hardware interfaces nor other processors tell a processor what to do; rather, tasks to be done are posted in the PID and processors ask the PID what should be done next.

There are some costs to such a passive system: First, the resulting slower responsiveness has necessitated additional buffering in some of the interfaces. (A side effect of this need for buffering is that more efficient interface design is possible because the buffering eases the timing constraints imposed on the

interface.) Second, the program must regularly break from tasks being executed to check the PID for more important tasks. The alternatives, however, are far worse. In a more active system, for example one which uses classical priority interrupts, it is difficult to decide which processor to switch to the new task. The possibilities for deadlocks are frightening, and the general mechanism to resolve them cumbersome.

As a third example of simplicity, the entire system is broken into reliability subsystems which are parts of the overall system that verify one another in an orderly fashion. The subsystems are cleanly bounded with well-defined interfaces. They are self-contained in that each includes a self-test mechanism and a reset capability. They are isolated in that all communication between subsystems takes place passively via data structures. Complete interlocking is provided at the boundary of every subsystem so that the subsystems can operate asynchronously with respect to one another.

The monitoring of one subsystem by another is performed using timer modules, as discussed below. These timer modules guarantee that the self-test mechanism of each subsystem operates, and this in turn guarantees that the entire subsystem is operating properly.

4.1.2.2 Redundancy

Redundancy is simultaneously a blessing and a curse. It occurs in the hardware and the software, and in both control and data paths. We deliberately introduce redundancy in the hardware to provide reliability and promote efficiency, and it frequently occurs because it is a natural way to build things. On the other hand the mere existence of redundancy implies a possible disagreement between the versions of the information. Such inconsistencies usually lead to erroneous behavior and can persist for long periods.

There are several methods of dealing with redundancy. The first and best is to refer always to a single copy of the information. Otherwise, we must check the redundancy and explicitly detect and correct any inconsistencies. What is important is to resolve the inconsistency and keep the algorithm moving. Sometimes it is too difficult to test for inconsistency; then timers are used as discussed below.

4.1.2.3 Timers

There is a uniform structure for implementing a monitoring function between reliability subsystems based on watchdog timers. Consider a subsystem which is being monitored. Such a subsystem

is designed to cycle with a characteristic time constant, and a complete self-consistency check is included within every cycle. Regular passage through this cycle is therefore sufficient indication of correct operation of the subsystem. If excessive time goes by without passage through the cycle, it implies that the subsystem has had a failure from which it has not been able to recover by itself. The mechanism for monitoring the cycle is a timer which is reset by every passage through the cycle. (For instance, in the IMP system, there are both hardware and software timers ranging from five microseconds to two minutes in duration.) Another subsystem monitors this timer and takes corrective action if the timer ever runs out. To avoid the necessity for subsystems to be aware of one another's internal structure, each subsystem includes a reset mechanism which may be externally activated. Thus, corrective action consists merely of invoking this reset. The reset algorithm is assumed to work although a particular incarnation in code may fail because it gets damaged. In such a case another subsystem (the code checksummer) will shortly repair the damage.

The entire system consists of a chain of subsystems in which each subsystem monitors the next member of the chain[20]. Lower

---

[20] See Appendix A for a specific example.

subsystems provide and certify some important environmental feature used by higher level systems. For example, a low level code tester checksums all code (including itself), insures that all subsystems are receiving a share of the processors' attention, and guarantees that locks do not hang up. It thus guarantees the most basic features for all higher levels. These will, in turn, provide further environmental features, such as a list of working memory areas, I/O devices, etc., to still higher levels.

Before they can work together to run the main system, a common environment must be established for all processors. The process of reaching an agreement about this environment is called "forming a consensus", and the group of agreeing processors is known as the Consensus. An example of a task requiring consensus is the identification of usable common memory and the assignment of functions (code, variables, buffers, etc.) to particular pages.

The Consensus maintains and counts down a timer for every processor in the system in order to detect uncooperative or dead processors. This monitoring mechanism includes reloading the failing processor's local memory and restarting it. Reliance on the Consensus is vulnerable to simultaneous transient failure of

all processors. For many cases (as for example when all of the processors halt), a simple reset consisting of a one-second timer on the bus and a 60 Hz interrupt routine suffices.

For more catastrophic failures the machine must be reset, reloaded, and restarted using external means.

4.2 System Reliability Techniques

As discussed above, there are a number of techniques which are used with any Pluribus system to insure availability. However, as was also mentioned, no attempt is made with these techniques to guarantee that the system never makes an error. Rather, an attempt is made to minimize errors and to recover from errors, but not necessarily without the loss of any data. Thus, there may be times when the Pluribus-based message system will cough and sputter and soon recover (in the sense that the hardware and software are ready to continue running the program), but without additional mechanisms, some data may be lost. For instance, if a memory failed, the entire contents of that memory could be lost although the system would adjust and return to operation using the remaining memory banks.

If the use of the Pluribus-based message system is to be like that of most existing message systems of which we know, the

users will require that the system not lose data (or at least minimize data loss) even through system failure and recovery. There are a number of more or less traditional techniques to guarantee system reliability, and these are applicable to the Pluribus-based message system as well as any other system. In the remainder of this section we discuss some of these techniques [21].

The problem of system reliability can be divided into two convenient categories: the problem of recovering from a disk (presumably the medium for the file system) crash and recovering from a CPU crash or program error. The following makes the dual assumptions (valid we think) that a disk crash is much less likely than a CPU crash or program bug, and that most program bugs will not hurt the file system.

Suppose the message system has been processing user transactions all day and then there is a failure which results in the state of the message system and its file system being uncertain. Uncertainty is worse than certain knowledge that a particular file was destroyed. If it was known for certain that

[21] The techniques in this section are for the most part proven techniques which are a part of the folk-lore of computer technology. The write-up here is, to our knowledge, a first. We wish we had had it when we began researching this area.

a particular file was destroyed and no other, then all but the user of the destroyed file could continue work. However, when the damage is uncertain, no user can continue work. Thus, something must be done to return the system to a known state. Actually, systems are frequently continued from the point of failure on the hope that not too much damage was done, and the thought that if any user (someday) detects that some of his data was lost, he may somehow be able to recover on his own. Such systems usually do dump the entire disk once in a while so that if it is completely apparent that the file system is lost, it can be restored to the point of the last dump, and the users only have to redo all their work from this point.

However, for our purposes a method which has less uncertainty and which puts less burden on the user is clearly required. The uncertainty can be removed by returning to the state of the system at the point of the last dump, each time there is a failure. (We will return shortly to the issue of lessening the burden on the user.) Checkpointing is the conventional name given to the process of taking a periodic dump of the system as mentioned above, i.e., periodically saving sufficient information to permit the system to be restarted at the previous point at which information was saved. The points

are called checkpoints[22].

There is an interesting theoretical issue (with much practical import) about checkpointing, namely the optimum checkpoint interval. The loss due to failure can be reduced by checkpointing often. However, it is expensive to save the state of the system too often. This problem of finding the optimum checkpoint interval has been studied previously[23].

We now address the problem of lessening the burden on the user. If, in addition to making periodic checkpoints, the system writes every transaction coming into the system onto a magnetic tape (the most sensible medium for this sort of job), then after a failure and backup to the checkpoint, the transactions which were processed after the checkpoint can automatically be replayed through the system for the users, thus relieving them of the burden of recreating their work themselves. This "log" of transactions is frequently called an audit trail (and can also be used for audit, accounting and other purposes). Clearly, a

---

[22] IBM Corporation, OS Advanced Checkpoint/Restart, Release 21.7, IBM Manual GC28-6708-5.

[23] Chandy, K.M. and Ramamoorthy, C.V., "Rollback and Recovery Strategies for Computer Programs", IEEE Transactions on Computers, Vol. C-21, No. 6, June 1972, pp. 546-556.
Young, John W., "A First Order Approximation to the Optimum Checkpoint Interval", Communications of the ACM, Vol. 17, No. 9, September 1974, pp. 530-531.

tradeoff similar to that mentioned above exists for a system which uses periodic checkpoints and an audit trail to minimize loss in the event of failure and to automate recovery. Checkpointing too often is wasteful but minimizes the cost of replaying transactions from the point of the last checkpoint. Checkpointing too seldom does not increase cost in and of itself, but results in having to play back too many transactions in the case of failure. This tradeoff is illustrated in Figure 7 (adapted from[24]). The curve on the figure labeled "overhead to checkpoint" indicates that if checkpoints are made very frequently, there is great overhead. As the inter-checkpoint interval increases, the overhead due to checkpointing decreases rapidly. On the other hand, there is little overhead due to having to replay transactions if checkpointing is done frequently; but this increases linearly (given a few reasonable assumptions) with the increase in the inter-checkpoint interval. Summing the two curves to find the total overhead for varying values of the inter-checkpoint interval, one sees that there is an optimum (i.e., point of minimum overhead). This point is the optimum inter-checkpoint interval.

[24] Chandy, K.M. et al., "Analytic Models for Rollback and Recovery Strategies in Data Base Systems", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 100-110.

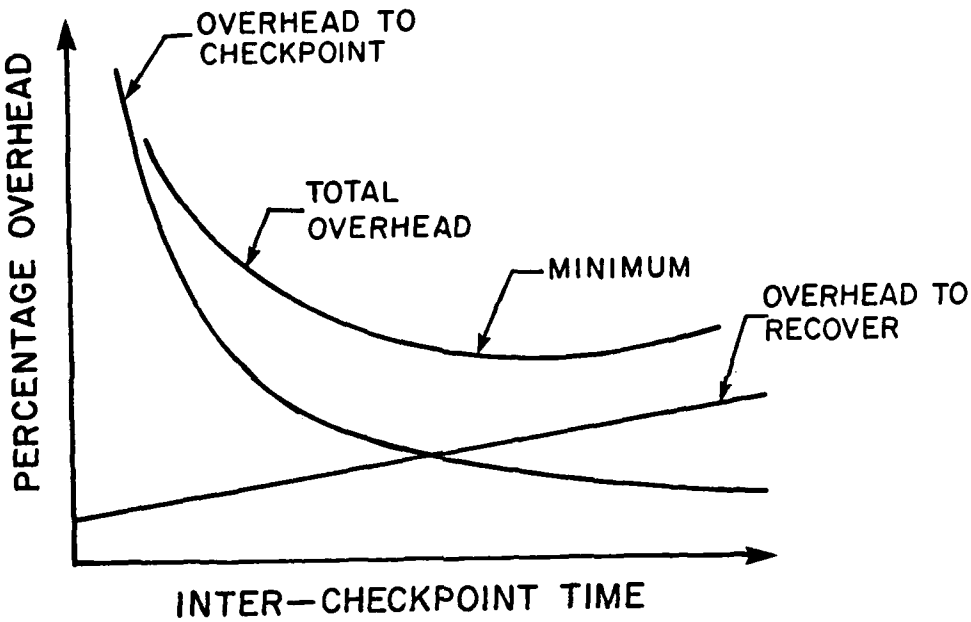


Figure 7 Intercheckpoint Ti... v. Overhead

Notice that while there is an optimum checkpoint interval, the simple approach of periodic checkpoints and replaying the audit trail is possibly still quite expensive. Each time the full disk is dumped, for instance, a great cost is incurred (e.g., half an hour for typically sized disks with typically fast tape drives); further, to reload the disk from the checkpoint is equally expensive. However, there is a technique for eliminating much of this overhead, at least in many cases. This technique involves keeping a log of before images. Assume for the moment that the disk is functioning well, but the CPU fails or the program fails because of a bug. Presumably, before the CPU or program failure, the disk was correct. After the failure, however, one is not sure of its state; it may not have been updated correctly to account properly for the transactions being processed at the time of the failure. Suppose in addition to checkpointing the entire disk each day (say) and keeping an audit trail of all incoming transactions, the following steps are taken:

a. All transactions into the system are logged as part of the audit trail.

b. Before any record is changed or deleted on the disk, the before image (the value of the record before the

change or deletion) is written on the (logically separate) checkpoint tape.

c. Periodically (and often, e.g., every five minutes) a modified checkpoint is written on the checkpoint tape. This modified checkpoint does not include the complete state of the disk, but rather includes only the complete state of the program (e.g., registers, buffers, swapping memory) and a pointer to the current position of the audit trail tape.

d. In the event of the failure mentioned above, the checkpoint tape is played backwards from the end[25], restoring disk records changed or deleted since the last checkpoint to their before images.

e. When the most recent checkpoint is reached, the state of the system is restored and the audit trail tape is rewound to the point at which it was at the last checkpoint. The system is now completely restored to its condition at the the time of the last checkpoint, including the state of the disk.

---

[25] Finding the end of the tape can be an interesting practical problem; typically one fills the blank tape with end-of-file marks to facilitate tnis.

f. Now the audit trail tape may be read forward, reprocessing transactions which arrived since the last checkpoint.

Thus, the addition of before images provides a capability to restore the state of the disk with very little overhead as compared to a total disk reload. Further, because the modified checkpoint is so small compared to a full disk dump, one can afford to checkpoint often. The result is low system cost for recovery and high user convenience.

There is a potential problem with the above system, the problem of order-dependent outputs. Suppose, for instance, in a banking transaction system, there are two parallel processes, one processing deposits and one processing withdrawals. Further suppose that for a given customer a withdrawal is processed which results in an overdraft situation and a nasty letter to the customer. Next the deposit process runs, and a deposit is processed for the customer which erases the overdraft situation. Then the system crashes and recovery is initiated. During recovery, because of process overlay considerations (for instance), the customer's withdrawal and deposit are processed in the reverse order and this order is written in the bank's records. When the user calls to ask about his overdraft, the

bank will have no record of it. In this particular case, there are constraints one could have placed on the way the system operates to eliminate the problem; however, in general, if one permits parallel- or multi-processing, there is no way to avoid such problems. One must either ask the users to insure that such problems cannot happen or can be corrected at user level, or one must forbid multi-processing (at a potentially great decrease in system efficiency).

This brings up the related "problem" of duplicate responses. Since one goes back in time during recovery and reprocesses inputs, outputs already sent to the user can result. It is best to declare such duplicate outputs to be a system feature, assuring the user that his transactions were, in fact, processed. Some systems attempt to avoid such duplicates by writing system outputs on the audit trail (along with inputs) and then matching outputs resulting from reprocessing against outputs on the audit trail and suppressing outputs which have already been sent to the user once. One can alternately use time stamps on the outputs to eliminate those already sent to the user before the system crash. There are two problems with trying to eliminate duplicate outputs: 1) one can never be completely sure all duplicates are eliminated; and 2) writing outputs on the audit trail can be

very expensive if the users require prolific output. Of course, it may be desirable to record outputs for other reasons, such as for accounting or statistics, or simply for the purpose of maintaining a complete output log.

In the above discussion we implied two separate tapes, one for the audit trail tape and one for the checkpoint tape. In fact, it is possible to use the same physical tape drive for both these functions, and it is conceptually quite straightforward to do so. One first works one's way backwards down the tape, restoring the disk from the before images until one reaches the last checkpoint which is used to complete restoration of the system to a known state. Then one reads forward on the tape, replaying the transactions. Using the same tape does slightly complicate the recovery software which must separate the two logical tapes, but this is certainly worth doing when compared to the cost of requiring two physical tape drives always operational.

With each record on the disk, one can maintain a field stating the date and time the record was last written (as will be shown below, this field is convenient to have for other purposes also). Just before a given record is to be changed on the disk, the record must be read into memory prepatory to writing the

before image on the checkpoint tape. By looking at the time the record was last written, one can determine whether a before image has already been written for this record since the last checkpoint (i.e., if the record has been written since the last checkpoint, then a before image has been written since the last checkpoint). If a before image for this record has already been written since the last checkpoint, there is no need to write another before image for the record, even though the record may have been updated several more times since the before image was originally written since the checkpoint. The additional before images for the same record need not be written since steps d. and e. above have the effect of restoring the disk to its state before the first time the record was updated since the checkpoint. Since it seems likely that a given record would be subject to repeated updates, closely spaced in time, this trick can result in considerable savings in not having to write redundant before images.

Now, suppose the disk itself crashes. With the techniques we have discussed so far, there is no alternative but to go back to the last complete disk dump, to reload, and to replay all the transactions since the complete dump.

Actually, some systems have used after images to attempt to minimize the expense of the recovery process in the case when the disk has crashed. After images is the name given to copies of disk records written to the checkpoint tape after the records have been updated (rather than before as with before images). When the disk crashes and one has after images available, after the disk has been completely reloaded from the point of the last disk dump, the disk can then be more quickly restored by simply updating disk records from the after images (in the order written) until the point of the last checkpoint is reached. From this point one may read the audit trail tape forward as in point f. above and finish restoring the state of the system. However, after images have several problems: 1) there are a lot of them since one must be written every time a record is written even if the same record is written many times; 2) they are written on the tape in the wrong order for convenient reloading (that is, they should be sorted to eliminate duplicates and to reduce disk seek time during reload); and 3) their being written takes processing time which increases proportional to system activity rather than being schedulable for some period off prime time.

A good alternative to after images for minimizing the cost of recovering from a disk failure is the incremental dump. With

the incremental dump, one periodically scans the entire disk looking for records which have been changed since the last incremental dump and dumps them. The same field in the records can be used to decide whether or not a record should be included in the incremental dump as is used to keep track of the date and time that records were last written, to reduce the number of before images written. In some cases it is more practical to keep a (sorted) list of updated records in core, or (in even fewer cases) to keep a chain of updated records on the disk.

As with the after images, a full dump must be made once in a while to serve as a base upon which to load the incremental dumps (or the after images in the previous case). When the disk fails, one first reloads the disk from the last complete dump. Then one reloads the incremental dumps since the last complete dump. Then one replays the audit trail tape from the point of the last incremental dump.

The incremental dump has several possible advantages over the method of after images: 1) the incremental dump can be done at a convenient time off prime shift; 2) the incremental dump includes at most one copy of a given record for each period between incremental dumps; 3) since one usually simply scans

completely across the disk to detect which records should be included in the incremental dump, an incremental dump tape can be reloaded without excessive movement of disk heads and is thus much faster. There is also an indirect benefit of having to scan the entire disk periodically for the purpose of the incremental dump: during the scan one may detect problem areas on the disk. If one goes too long between complete scans of the disk, a portion of the disk may develop unrecoverable trouble before one detects that there is trouble. Once per day is a natural and often-used frequency for incremental dumps. However, if one wants to save the complete scan, especially if the portion of the disk updated in the period between incremental dumps is a small fraction of the total size of the disk, one may keep a table of the records changed and use this table to select the records to be incrementally dumped. Even with this table, one can still dump the records in a convenient order to reload.

In addition to saving reprocessing of transactions, incremental dumps (and to a lesser extent after images) permit one to lengthen the time between full dumps. This can be a significant saving. If there are many incremental dumps to be reloaded after a failure, one might think of using read/merge techniques.

In some applications one can not afford the down time to reload the complete dump and the incremental dumps. In such instances, one must have two parallel disks, each of which is updated identically and in parallel. Then if a disk fails, one simply plays back through the before images and the last checkpoint and back forward through the transactions to get the system back to the point of the failure and continues using one disk. Of course, a second disk also means the elimination of the down time which would otherwise result while the broken disk is repaired.

In all of the above techniques, care must be taken to properly interlock checkpointing, dumping, the arrival of transactions, and the recovery process. For instance, while the system is recovering, new incoming transactions must be held off, either by forbidding them completely, or by recording them for later processing once the system has recovered. Traditionally, while a system is dumping, new transactions have also been held off. Actually, it is possible to avoid the loss of system availability during dumps. First, one arranges that the dump time correspond with the time of a checkpoint (which is necessary in any case). Then during the period of the dump, the before image for any record updated is written on the dump tape as well

as on the checkpoint tape. If the system later has to be restarted or the disk restored, all the information needed for this to occur using the standard recovery techniques is already on the dump and checkpoint tapes.

4.3 Availability through Distributed Computation

In a network environment, such as that in which a PMS might reside, it becomes possible to consider enhancing the availability of the system through distributed computation techniques. For instance, when there are several PMSs attached to the same network, if a particular one goes down, its users might temporarily use another on the network, even though their own message system will soon recover and recover reliably. Admittedly, this is a simple form of distributed computation.

There are several areas in which distributed computation might help the PMS application. These are discussed in the following four subsections.

4.3.1 Simple Complete Backup

This is essentially the technique mentioned in the previous paragraph. When a user's own PMS goes down, he explicitly and manually begins to use an alternate PMS. The PMSs are

independent and no attempt is made to automatically take work begun on one system and continue it on the other system. This is analogous to the computer user who has two computers available and is running a time-consuming Fortran program on one of them. When that first system halts, on a breakpoint for instance, rather than going out for coffee, the user might do a little Cobol work on the other computer, with the hope that the first computer will eventually recover and continue his Fortran job from where he left off.

Owners of independent PMSs might make arrangements with each other to provide each other with backup service for high priority users.

Obviously, if complete backup is a good idea, partial backup is also useful where possible. For instance, if the PMS consists of two parts, the handling portion and the terminal handler, it might be possible to backup the terminal handler with another terminal concentrator also on the network.

4.3.2 Use of Selected Network Resources

Aside from providing a backup capability for increased availability of all or part of the PMS, a network also offers the possibility for use of certain selected resources not otherwise

available or which the PMS would otherwise have to provide itself. For instance, a PMS on the ARPANET might make use of the DataComputer rather than supplying that much rapid access archival storage at the PMS itself. It is easy to envision ways to take convenient advantage of a DataComputer-like device on the same network as a PMS. For instance, the complete dumps of the disk might be made to the DataComputer, although unless the network were extremely fast, the process of dumping and reloading would be very slow. More likely, one could simplify the construction of the PMS by supplying a relatively modest message storage space for each user. Then either the system (automatically) or the user (explicitly) could archive a given set of messages on the DataComputer. Then when the user wanted the messages back, they would be retrieved from the DataComputer and restored to the user's message storage space local to the PMS. One might ask, why not just archive to magnetic tape. The answer is that for limited size transactions, e.g., a file of messages on a particular topic, the retrieval time from the DataComputer is likely to be less than the time required to mount and scan the tape. This is in contrast to the case where a relatively large amount of data is to be retrieved, when it would be faster to find the tape, mount it, and read all the data from that tape rather than incurring the relatively slow transfer of

all that data across the network. Obviously the tradeoff is transfer rate vs. seek time and the correct solution is a function of the frequency of seeks.

Another example of selected use of network resources would be to not have a terminal handler on the PMS at all but to assume that all terminal handling is provided elsewhere in the network.

4.3.3 Distributed Data Bases[26]

If one attempts to provide availability through use of distributed computation and to provide it reliably (e.g. transparent to the user and without errors), one is faced with the problem of maintaining distributed data-bases. A distributed data base might also have advantages in the areas of increased responsiveness and load sharing potential. For instance, data base queries initiated at sites where the data is stored can be satisfied directly without incurring the delay due to transmission of the queries and responses throughout the network, and those initiated from sites "near" the data base's sites can be satisfied with less delay than those further from the data base sites; with regard to load sharing, the computational load

[26] R. H. Thomas, "A Solution to the Update Problem for Multiple Copy Data Bases", submitted for publication.

of responding to data base queries can be distributed among a number of data base sites rather than centralized at a single site.

These and other benefits of replicating data must be balanced against the additional cost and problems introduced in doing so. There is, of course, the cost of the extra storage required for the redundant copies. There is the problem of maintaining synchronization of multiple copy data bases in the presence of update activity. Other problems are determining for a given application or subapplication the number of copies to maintain and the sites at which to maintain them, selecting a data base site to satisfy a query request when it is initiated, etc.

The inherent communication delay between sites that maintain copies of a data base makes it impossible to insure that all copies remain identical at all times when update requests are being processed. The goal of an update mechanism for a multiple copy data base is to guarantee that updates get incorporated into the data base copies in a way that preserves the mutual consistency of the copies in the sense that all copies converge to the same state and would be identical should update activity cease.

Traditional update mechanisms can be characterized as involving some form of centralized control whereby all update requests are channeled through a single central point. At that point the requests are first validated and then distributed to the various data base sites for entry into the data base copies. A second, fundamentally different approach to the update problem, based on distributed control, is possible. For this approach the responsibility for validating update requests and entering them into the data base copies is distributed among the collection of data base sites.

Mechanisms which use centralized control are attractive because a central control point makes it relatively easy to detect and resolve conflicts between update requests which, if left unresolved, might lead to inconsistencies and eventual divergence of the data base copies. The primary disadvantage of such mechanisms is that data base update activity must be suspended whenever the central control point is inaccessible. Such inaccessibility could result from outages in the communications network or of the network site where the control point resides. Because a distributed control update mechanism has no single point of control, it should, in principle at least, be possible to construct one which is capable of processing data

base updates even when one or more of the component sites are inaccessible. The problem here is that it is non-trivial to design a mechanism which can resolve conflicting updates in a way that preserves consistency of the data base copies and is deadlock free. Centralized update control is adequate for many applications. However, there are data base applications whose update performance requirements can be satisfied only by a system which uses distributed update control.

An algorithm to correctly support distributed data bases should have the following properties:

- Distributed updating. Updates to a redundantly maintained data base can be initiated through any of the data base sites.

- Update synchronization. Races between conflicting, "concurrent" update requests are resolved in a manner that maintains both the internal consistency and the mutual consistency of the data base copies.

- Deadlock prevention. The synchronization mechanism that resolves races does not introduce the possibility of so-called "deadly embrace" or deadlock situations.

- Robustness. The data base update algorithm can recover from and function effectively in the presence of communications (network) and data base site (host) failures. The algorithm is robust with respect to lost and duplicate messages, the (temporary) inability of data base managing processes to communicate with one another (due to network or host outages), and the loss of memory (state information) by one or more of the data base managing processes. In developing the algorithm, any mechanism that required all data base managing processes to be up and accessible in order for it to function effectively should be rejected. A mechanism should be sought that requires only pairwise interactions among the data base managing processes.

4.4 User Level Reliability Techniques

Despite everything the system can do to provide reliability, users will also want user level reliability mechanisms, either because they do not trust the system[27], or because they do not

[27] For a discussion of the environment users may face, see J.M. McQuillan and D.C. Walden, "Some Considerations for a High Performance Message Based Interprocess Communication System," Proceedings of the ACM SIGCOMM/SIGOPS Interprocess Communication Workshop, March 1975, pp. 77-86.

trust the user at the other end. We look at the analogy of the U.S. Mail (although the analogy is admittedly weak in the area of the assumption that the system is doing its best to be reliable). After one mails an important letter, one often calls the party to whom the letter was mailed to make sure he received the letter. Alternatively, sometimes when one mails a letter, one requests a receipt from the receiver to make sure he accepted the letter. The message system should make provision for this sort of user level reliability mechanism. For example, the following options should be available:

- Automatic receiver acknowledgment of receipt of message
- Sender requested receiver acknowledgment of receipt of message
- Special handling -- special end-to-end system acknowledgment and retransmission of message
- User level logging of message receipt and transmission.

5. SYSTEM SECURITY METHODS

In some environments, it is quite important to assure that the messages processed by the system are safe from disclosure to unauthorized persons. In a secure military environment, for example, the importance of this assurance is great enough to warrant explicit protection techniques even though they may increase the system cost appreciably.

The designer of the secure system is faced with a seemingly unlimited variety of security hazards; Section 5.1 below and its subsections detail a number of them. These security hazards range from human errors on the part of the users to subversion during the design and implementation of the system or even after its installation. Between these extremes we find that many simple and likely hardware failures and software flaws are potential security hazards. Although we have not ignored the security hazard caused by user error or subversion, we have concentrated our efforts during the study on the latter group of hardware and software faults, since they seem better suited to technological solutions.

During the course of this study we have developed methods for configuring the Pluribus computer into a message system and

techniques for processing messages which permit secure operation in the face of either hardware failure or software flaws. These techniques are well suited to the PMS environment. Various combinations of them can be applied at a reasonable cost to provide the desired level of security. These techniques take advantage of the flexibility of the Pluribus to configure a secure system with no change to the basic machine, the only changes being to the interface I/O system. The objective of these techniques is to prevent the delivery of messages to unauthorized users. We expect to be able to assure this even in the face of any single failure or software design flaw (bug). In the case of hardware failure, we expect to be able to provide rapid detection of any failure which leaves the system in a degraded state. As a result of this study, we can meet these objectives in an efficient and modular manner with minimal increase in cost. The techniques which achieve these objectives are described in detail later in Section 5.2 below and its subsections.

5.1 Security Hazards

Our discussion of security hazards is divided into six subsections: the I/O system, the memory system, processors, software flaws, subversion, and user error.

5.1.1 Security Hazards -- I/O System

Security hazards in the I/O system include:

1) misdirection of a valid message to the wrong I/O device;
2) copying a message on a second terminal during a normal I/O transfer; and
3) transferring the wrong part of memory to the I/O device.

These hazards are caused by failures in device address logic, memory address logic, or the address paths to I/O devices or memory.

Figure 8 shows a common structure for computer I/O systems. In the simplest structure based on passive, polled I/O devices, each I/O device is physically separate and is attached to a device address bus and data bus. The processor accesses the I/O devices by presenting the device address on the device address

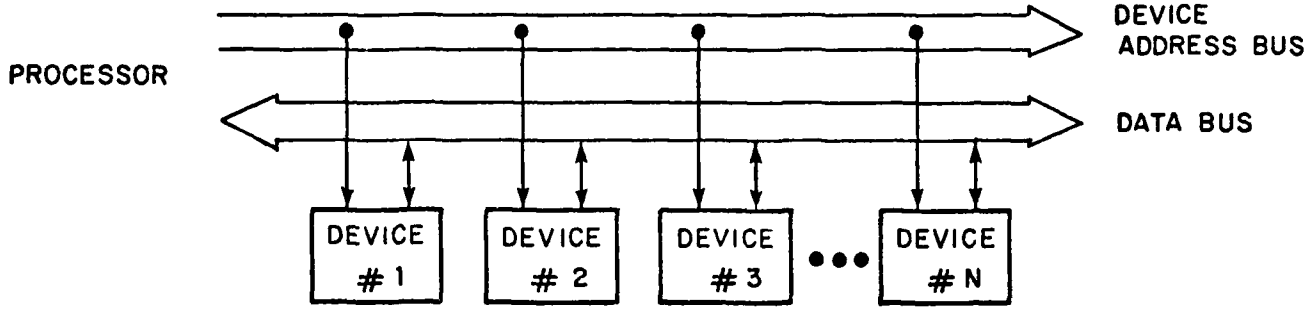
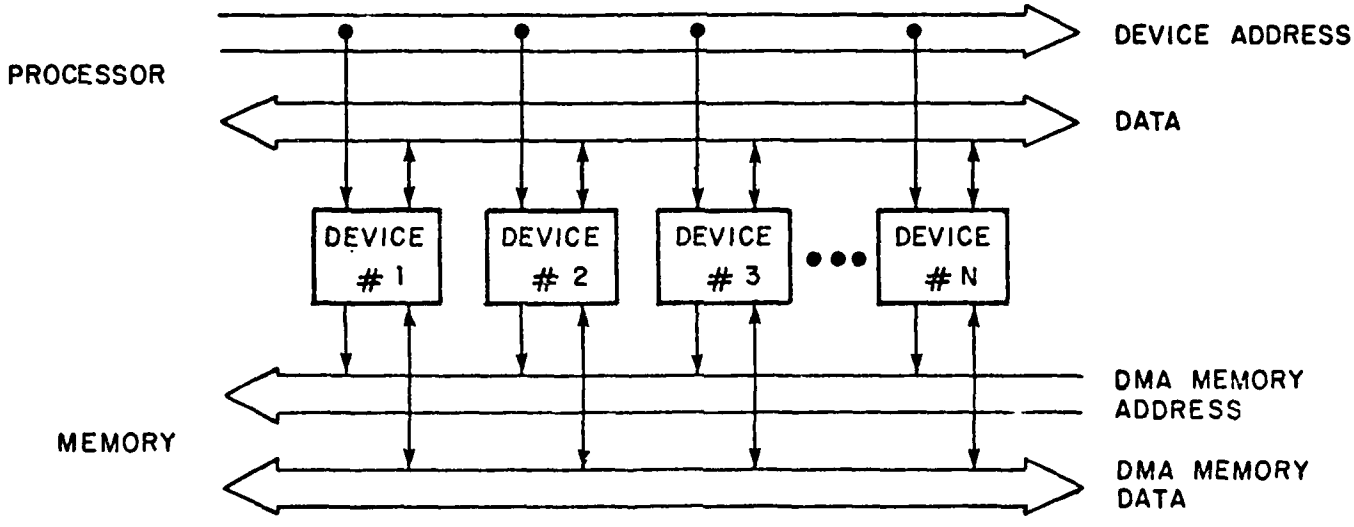


Figure 8 Polled I/O System

bus. The device recognizes its address and either accepts data from the data bus or presents data to the data bus. In this simple environment we can see how the first two I/O system security hazards can arise. Misdirection of text to the wrong I/O device can arise because of bit errors in the address bus. Since this is typically a bus which drives many I/O devices, a failure in any I/O device which shorts an address wire to ground can cause this type of failure. Similarly, if a bit failure occurs in the address recognition logic of one of the devices, that device may accept transactions intended for some other device. This can produce an extraneous copy of the message.

The structure of a direct memory access (DMA) system is shown in Figure 9. The top portion of this system is the same as for polled I/O and provides the mechanism by which the processor communicates with the I/O devices. The direct memory access feature is achieved by adding a memory bus which the devices use to transfer data directly to or from memory (thus achieving a high data rate while relieving the processor of the task of polling). Bit errors on the DMA memory address bus or the processor data bus can result in sending the wrong portion of memory to a device.



DMA I/O

Figure 9 DMA I/O System

In addition to these weaknesses, these types of I/O structures have some strong points with regard to security. Primarily, there is a reasonably high degree of physical separation between devices as illustrated in Figure 10. This physical separation makes it extremely unlikely that other hardware failures within the device could become security hazards.

Having presented several failure modes for the I/O system of a computer system, we come to the key question: What alternative protection mechanism can be implemented against the possibility of these failures? Three solutions are apparent:

1) classic error control (e.g., parity) techniques applied to all address paths, address holding register and address recognition circuits throughout the system;
2) full redundancy and separation in all of the address paths and logic in the I/O system; and
3) data segment oriented error control.

Classic error control techniques, such as parity, have been used for many years in computer systems to detect data errors. In this system, however, they would be applied to the address paths, since for security purposes, we care much more about

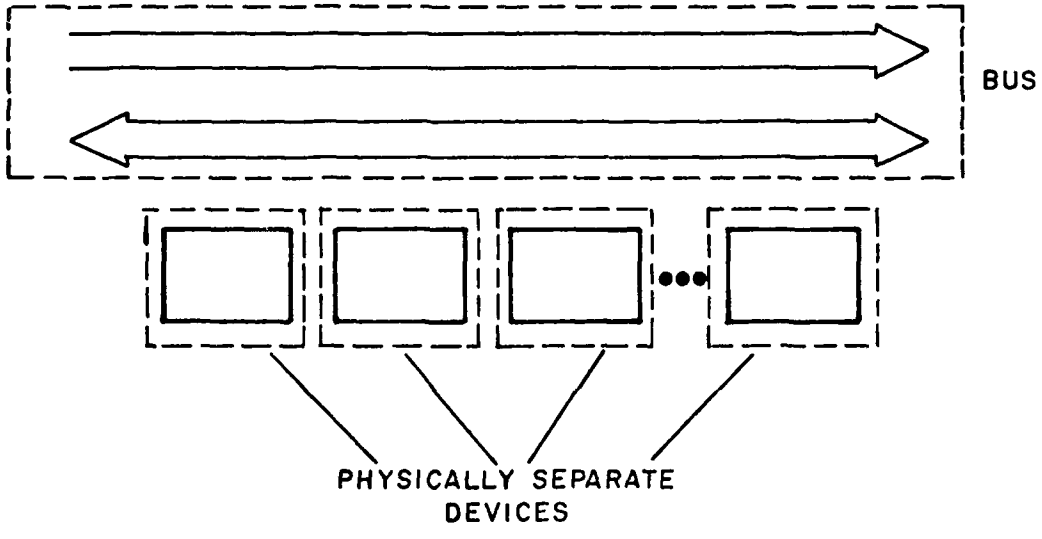


Figure 10 Physical Separation of Devices

addressing errors than about data failures in the I/O system. (The correctness of the data path which selects and specifies the address of a buffer is also quite important to the security of the system.) There are two major problems to this approach: First, simple parity is probably not adequate protection against these types of faults since it detects only an odd number of bit errors. Many faults (i.e., the failure of an IC which drives four address lines) can produce an even number of bit errors which would then be undetected. Second, this technique must be scrupulously applied throughout the entire address structure of the machine, which most minicomputers (including the SUE) do not do. In either case we believe that one of the other approaches must be used.

Figure 11 illustrates the second solution. This approach uses two independent interfaces for each I/O device, serviced by independent I/O busses. A comparator is used to insure that the outputs of the two interfaces are suppressed if not identical. This approach still suffers from the weakness that if it is being serviced by a common processor or memory, an address failure in that processor or memory represents a security hazard. This approach is also quite expensive since it duplicates so much hardware. A critical component in this type of design is the

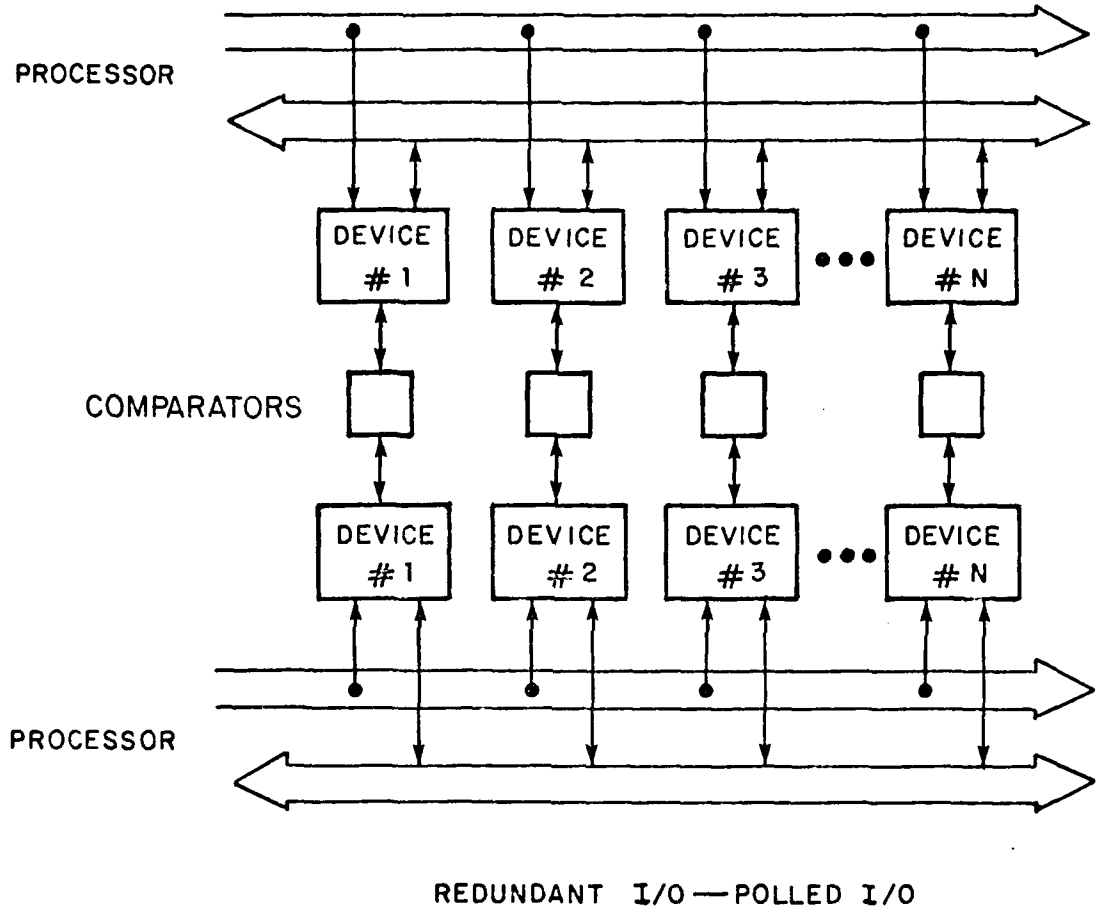


Figure 11 Fully Redundant, Fully Separate I/O System

comparator. It is important that it be tested periodically so as to insure that any failure in it is rapidly detected.

Before discarding this approach, we should note that if the two independent I/O systems interface to two independent processor and memory systems, secure operation is achieved.

The third approach is illustrated in Figure 12. In this approach errors are controlled by a multiple bit checksum on each segment of data. Each data input or output takes the form of a block consisting of a device number, text, and a checksum which includes both the device number and text. Each I/O device has the necessary hardware to accept and buffer one of these blocks while verifying the checksum and device number. The device would output the text if and only if the device and checksum were valid. On input, the device packages the incoming data in the same format and attaches the device number and checksum. With this approach, errors in the address path do not represent security hazards since the device address is part of the data itself and is checksummed. The device number must be included in the segment because it is necessary to define the physical I/O port which is to transfer this segment. The device number was chosen instead of, for example, the user name, so that it will be

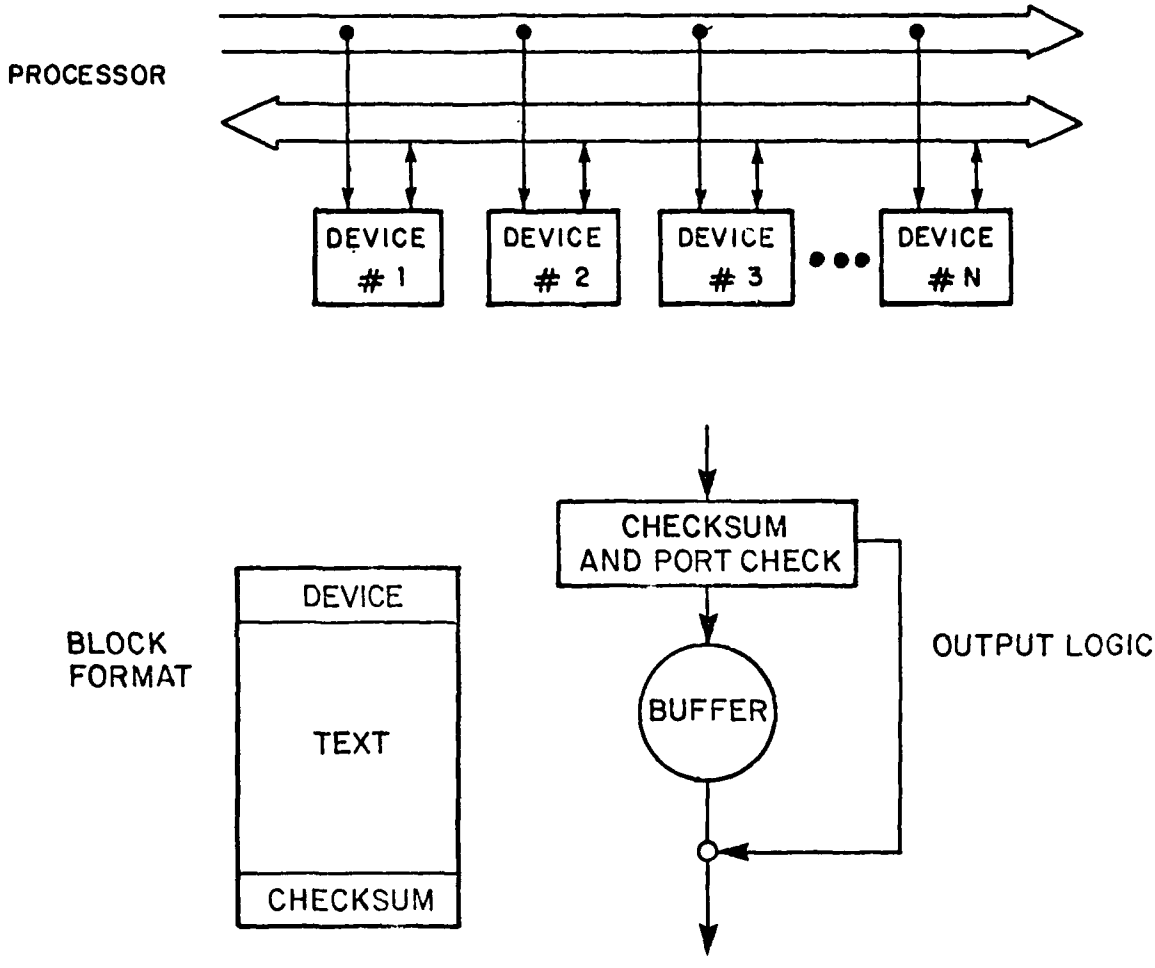


Figure 12 Data Segment I/O Error Control

easy for the device to recognize whether the field is correct. Notice that this technique provides protection against address failures in the memory and processors as well as the I/O system itself. The disadvantage of this approach is that buffering for the entire text segment, checksum logic and device number checking logic is necessary in every I/O device. However, it is more efficient than doubling all of the interfaces, as suggested in the previous approach.

5.1.2 Security Hazards -- Memory System

Security hazards in the memory system are address errors or data errors -- they can occur on instruction fetches or data fetches (errors in writing appear only when read). If the error occurs on an instruction fetch, it modifies the algorithm. If it occurs on a data fetch, it can result in incorrect pointers to message or other data structures.

These security hazards in the memory system are similar to some of the hazards in the I/O system and illustrate that a successful strategy must protect against errors in both the address and data portions of the system. These failures in the memory system are also similar to common software flaws such as accidentally referencing the wrong data structure or accidentally

writing to the wrong place in memory because of a bug in handling pointers. Because we would like to deal with hardware failures in the memory system as well as these types of software problems, error control on the hardware in the memory system is less attractive than checksums on blocks of data and redundant, physically separate, copies of the critical elements of the data.

These security hazards in the memory system also point out a weakness in the various "capability" mechanisms[28] normally used with security kernels. It does little good to prevent the software from accidentally accessing the wrong data structure if the memory system itself may fail so as to produce the same accessing error.

If error control is used to cope with these types of security hazards, it is clear that normal parity (which only includes the data stored in memory) is not adequate because of the possibility of serious errors caused by address bit failures. A straightforward solution to this problem is to include the memory address in the parity calculation as is done in the Pluribus. With this type of address and data parity, whenever a word is written into memory, the parity bit which is written with

[28] B.W. Lampson, "Dynamic Protection Structures," AFIPS Conference Proceedings 35, November 1969, pp. 27-38.

that word is the exclusive-or of all the address and data bits. When the word is read from memory, the parity bit is checked against the exclusive-or of the address bits of the address that was requested and the exclusive-or of the data retrieved from memory. Therefore, if the memory system accesses the wrong address because of a single bit error in the address logic, the parity bit retrieved will be incorrect and the fault will be detected. This approach still, of course, has the same weakness as any parity scheme, namely that there is only one bit of redundancy and thus only single bit errors are reliably detected.

The two methods which seem most attractive for assuring secure operation of the memory system are checksums on all of the important data in the system as was described in solution three of the previous section and redundant execution of the critical algorithms in two separate and physically independent systems. We describe this solution in detail in section 5.2.2.

5.1.3 Security Hazards -- Processors

Processor (or processor bus) failures which produce security hazards are few, both because there are few paths and storage elements in a processor and because these paths and storage elements are constantly exercised by many parts of the algorithm.

As a result, a failure in a storage element or a data path in a processor will affect many portions of the overall system and will be detected rapidly. Processor failures which could affect the I/O system were described in section 5.1.1. Failures in the data path to memory or the address path to memory could be security hazards in the way described in section 5.1.2. Failures in the address path are much more serious because they are much more likely to go undetected for a longer period of time.

5.1.4 Security Hazards -- Software Flaws

Many software flaws are like the hardware failures described in previous sections in that they result in erroneous access to system memory or a transfer of data to the wrong I/O device. There seem to be three methods for dealing with software flaws:

1) the classic debugging cycle where flaws are detected and repaired one by one as they reveal themselves;

2) controlled software environments such as user mode and capabilities mechanisms; and

3) automatic or manual program verification.

The first approach is certainly simplest since debugging must be done anyway, but requires that every flaw be active

before it can be repaired. It suffers from the weakness that one can never be sure that the last flaw has been discovered.

Controlled software environments, whether produced by a full scale capabilities mechanism or a bi-modal user and supervisor mode mechanism, provide a reasonably high degree of protection for the code in the controlled environment. However, this approach has the disadvantage of requiring elaborate software structures to deal with the controlled environments (setting up the capabilities and communicating the data between various system modules). These structures are often large and are themselves vulnerable to system flaws which represent security hazards.

Automatic software verification is the most powerful means of assuring secure software. Unfortunately the state-of-the-art in software verification limits its usefulness in validating large systems such as we anticipate for the PMS.

5.1.5 Security Hazards -- Subversion

Subversion can occur during system design, implementation, or operation. The primary techniques to deal with it are not technological. They include:

1) government review
2) personnel screening
3) physical security
4) automatic verification.

5.1.6 Security Hazards -- User Error

The most common form of security hazard is user error. Although we do not currently envision any user programming in the PMS, it is still important to guard against security failures because of errors on the part of users. Two important classes of these hazards are addressing a message incorrectly and addressing it to a recipient who is not cleared to receive that class of message.

A simple cause of misdirected messages is an error on the part of the sender in specifying the recipients. For example, in typing in the address for a message, the sender may misspell the recipient's name; if that misspelled name happens to match some other user of the system, it will be difficult for the system to detect the error. For example, in sending a message to Allen Smith he may address it to SMITH, not realizing that the message address for that user is ASMITH. The method for dealing with this type of failure is positive identification of each recipient

(i.e., ASMITH - SMITH, Allen). Clearly some human engineering of this type of mechanism is important. The first step is to make it quite easy to use distribution lists for messages with safeguards so that distribution lists are unique and distribution list names are long enough to greatly reduce the possibility of error.

To assure against sending classified messages to users whose security clearance is below the level of the message, it is clear that a failsafe way of specifying the security of the level for the message must be implemented and the PMS must verify that each recipient is authorized to receive messages of that security level. The former is an administrative and human engineering matter while the latter is relatively easy to do with software.

5.2 Secure System Design

Three important system design themes were developed during the study. First, division of the Pluribus message system into two different software environments. In one environment the message parts are vunerable and the program is run in two parallel but independent paths which are compared at their conclusion. For the benefit of discussion, these two paths are called blue and green. The second environment exists for

processing complete and otherwise protected message segments. This is a single program thread and is discussed here as the orange path. The orange machine performs almost all of the message handling tasks. The other two machines, the blue machine and the green machine, operate in parallel and perform only those tasks which are critical to the security of the system (those which change checksummed data segments -- see below). Providing two machines to perform the critical portions of the message handling tasks makes the system immune to an extremely wide range of hardware failures.

Second, to establish secure operation in the primary machine, every message in the primary machine, every fragment of a message as it is being built up, and every important data structure is protected by a powerful error detecting checksum. Every message fragment also carries an access list. All operations which require a modification to the checksum or which perform input or output to users are performed independently by the security kernel machines. This assures detection of any hardware error in the primary machine through the error detecting checksum and access list and in the security kernel machines through independence. (Actually, as will be discussed in detail below, two checksums are used on every critical element.)

This checksumming technique, while permitting the separation of critical and non-critical code, nevertheless allows an efficient and cost-effective solution by minimizing the repertoire of commands which must be performed by blue and green. Furthermore, this separation simplifies the task of software validation, since the critical code is only a small portion of the system software and is effectively insulated from the non-critical code.

The third system design theme is redundancy in the I/O structure to assure secure operation in the face of hardware failures in the I/O system. Much as redundancy in execution by blue and green provides protection against security failures due to hardware failures during message processing, redundancy in the I/O system assures against security failures due to hardware failures in the I/O system.

These system design themes were developed primarily to provide protection against hardware failures and accidental software flaws. However, to assure a truly secure system, protection must also be provided against user errors and malicious activity. User errors can only be detected through careful human engineering of the critical portions of user interaction such as establishing the list of addressees for a

message. Because the PMS is a dedicated single use system, most of the problems associated with malicious users are avoided. For example, since users do not write any software for the machine there is no need to provide constrained environments for "user" code. The problem of malicious activity on the part of the software development personnel is best handled in this context by software verification where possible and careful review of the software development elsewhere. The techniques developed to cope with hardware failures and accidental software flaws also provide some protection against the actions of malicious programmers.

In short, we have developed a system design based on the Pluribus which assures secure message processing through a carefully engineered combination of proven techniques. These techniques, which will be discussed in the following subsections, may be summarized as:

1) checksummed messages
2) redundant execution
3) physical separation
4) secure I/O structure
5) software verification.

This system design has an inherently high degree of security. It provides mechanisms which assure that security failures will not occur through any single hardware failure or any single software flaw. Furthermore, to cope with multiple failures, hardware or software failures which lead to a degraded level of protection will be detected rapidly. The cost of providing this protection is reasonably low both in its impact on the throughput of the system and in the amount of equipment and software which must be implemented.

Although the first four methods are designed primarily to cope with hardware failures, they also provide a level of software security which makes security failures through software bugs extremely unlikely and detection of software flaws which would be able to leak secure data relatively certain. This combined with software verification of a small part of the code permits a cost-effective solution to the software certification problem.

5.2.1 Checksummed Data Units

Checksummed messages and checksummed data segments permit efficient implementation of security in the PMS. The basic principle behind checksummed messages is as follows:

Every message and message fragment and every sensitive data structure has a list of users with access rights to that data and two checksums (which we call blue and green) to insure the integrity and continuity of the data and access list. The orange machine never changes any of the data in these checksummed segments nor does it ever adjust a checksum on these segments. Furthermore, we take care to assure that an item without a valid checksum will not be permitted to leave the PMS. Since all I/O to users passes through the blue and green machines, a barrier can be established. If the blue checksum is not valid, the blue module will block the packet's exit; while if the green checksum is not valid, the green module will block its exit. If its addressee or security limits are incorrect, both the blue and green modules will block its exit. Since all of the data in the orange machine is protected by a checksum and the orange module never adjusts the checksum, it can handle all messages and segments of messages during normal data processing with assurance that if by accident some of the data in a segment is altered by orange, that alteration will be detected by the checksum.

As an example of the division of tasks and the communications between blue/green and orange, consider the actions during the generation of a new message. Characters enter

both blue and green from the user's I/O device. Blue and green perform the tasks of terminal echoing and monitoring for any special control characters. They gather together a number of characters into a data segment, and place the user name associated with that port on the access list for the resulting data segment. Then both blue and green calculate their respective checksum for the data segment. At this point, the operations performed by blue and green are slightly different in that the checksums that blue and green compute are different from each other. Orange (whose job it is to do most of the processing) has established a character input queue; blue or green (whichever gets there first) locks that queue to block access to it by the other modules. Assuming that blue gains access, blue then writes the entire data segment with its access list characters and the blue checksum into a buffer, places it on the input character queue and marks it as having been serviced by blue, and unlocks the character input queue. Green, either because it recognizes that it has some input for that terminal or because of a periodic attention to that queue, looks at the data buffer and finds that it has been processed by blue but has not been processed by green. It therefore locks that queue, adds the green checksum (calculated from green's internal buffer) to the data segment, marks it as having been processed by green, and

then unlocks the queue. Orange now wakes up (perhaps having been prodded by blue and green) and notices that a new data segment is on the character input queue and that it has been processed by blue and green. Orange processes these input characters to determine whether they represent control information or text and performs the appropriate processing. For example, assume that a message is being constructed and that these characters are the next characters of the text of the message and should therefore be appended to the message that is being compiled. Orange has been maintaining a copy of the message as it has accumulated so far, but must get help from blue and green to add these characters to the end. It does this by making up a block which indicates the operation to be done (in this case joining the two data segments together) and which contains pointers to both of these data segments and the destination buffer where the result of the operation should be placed. Orange then places this block on a queue for processing by blue and green. When blue and green find that this block is ready for processing, they copy the two segments into their own private memories and then check their respective checksums. If the checksums are valid, the processing can continue. If not, further processing of that request is blocked and an error message is generated. Blue and green then independently make sure that the requested operation is valid and

append the new characters to the text of the original message (in fact only one of them needs to perform this operation). The list of authorized users included in this new data segment is the intersection of the access lists for the two original data segments. Blue and green calculate their respective checksums, append them to the data segment and mark it as having been processed. One of them then moves it to the queue of operations processed by blue and green. Should blue or green make an error in performing the requested operation, the result will not be the same in the two machines, and the checksums will not cover the same result. This will be detected the next time the checksummed segment is given to blue and green. For example, if blue makes a mistake and performs an incorrect operation, and furthermore, if blue appends the wrong characters to the text of the origional message, the green checksum will detect the error since it is based on a copy of the message which has the correct characters appended to it.

These checksums on data segments are in fact quite useful in other contexts. For example in the file system (which is naturally likely to make some errors) these checksums permit error control. Notice that the file system can be entirely under the control of the orange machine since the checksums on messages

assure that errors in storage and retrieval of these data segments will be detected. This removes a large portion of the processing burden from blue and green.

In addition to manipulating checksummed message fragments, the blue and green machines must be able to perform certain operations on other types of checksummed data structures such as tables. As an example, we describe a possible implementation of the login procedure on this machine. Before the login the port is idle and no user name is associated with it. The user arrives at the port and starts transmitting characters that indicate a desire to make use of the system. As the characters arrive at blue and green, they are gathered into checksummed segments. Since the user name is as yet unknown, the port number associated with that interface is placed on the access list of that data segment to indicate where the characters came from. This checksummed fragment of text is passed to orange as described above; orange performs the functions of command interpretation. Orange sees that this is a new user who wants to log in and sends out a canned message asking for the user name and password. When the user types his name and password and the checksummed data segment finally reaches orange, orange looks up the user name in the user directory and finds the password associated with that

user. (The user name and password data base is here assumed to consist of a large number of short checksummed data segments each containing a few user names, characteristics and passwords.) The objective is for orange to convince both blue and green to associate this new user name with that particular port number. Orange passes a pointer to the correct section of the user name and passwords data base and another pointer to the checksummed data segment which contains the user name and password typed in by the user. Blue and green now have the appropriate information and can verify that the correct user name and password pair has been presented and that the password table segment is valid, and can bind that user name to that port.

This sequence illustrates one of the keys to efficient system design: a technique which minimizes the memory and processor requirements of blue and green. The orange machine takes care of all of the protocol and maintains the user name and password data base. This saves a large amount of memory and processor bandwidth in blue and green. Furthermore, the various error routines and other aspects of user dialog are implemented only in orange. It is fair to say that orange does all of the work and passes the results to blue and green for approval.

This approach of physical separation between the critical and non-critical portions of the machine together with checksummed data segments in the non-critical portion of the machine offers protection against hardware failures and software faults while providing a lesser degree of protection against malicious code in the non-critical portion of the machine. Since every critical element of data in the orange machine is protected by a checksum, single hardware faults in the orange machine do not represent security hazards. While a hardware failure may alter the text of the message, it will be unable to preserve a correct checksum. It is conceivable that a hardware or software flaw in orange could produce a correctly checksummed segment with bad data. However, it is extremely unlikely, particularly since there is no code in orange which calculates checksums. Such code is totally unlike other code, and the chance of a flaw duplicating this code is vanishingly small. Similarly, a software flaw in the orange machine may erroneously present the wrong piece of data for an operation; however, the checksum and access list will permit blue and green to detect the error.

We expect the code in blue and green to be quite small and simple and reasonably easy to certify, whether through automatic software verification or manual techniques. The orange code, on

the other hand, is quite large. It includes the file system software, editors, command interpreters, and the network protocol modules. Because of its size, this software will be difficult to verify given the state of the art of software verification. As software verification techniques are improved, however, verification of this portion may become practical. At that time, we will have technological insurance against malicious activity on the part of the system programmers. Until then, the techniques of personnel screening and over-the-shoulder review can be used to provide the best available assurance of secure software.

5.2.2 Redundant Execution

Redundant execution permits detection of a wide class of hardware failures. By redundant execution we mean execution of the same algorithms on two separate sets of hardware.

In its simplest, most brute force implementation, redundant execution would have two copies of the entire message system. All inputs to the message system would be given to both systems. All outputs of the message system would be generated independently by the two systems and would be compared against each other. The result would be passed on to the user only if

the outputs from both of the machines agreed. This achieves a high degree of protection against hardware failures since if a hardware failure occurred in one machine which would cause a leakage of secure data, that leakage would be detected unless the same or a corresponding hardware failure occurred in the other machine as well. Unfortunately, this approach has the disadvantages of 1) high cost because all of the equipment must be doubled and further comparison logic must be added, and 2) high technical difficulty because of problems with synchronization between the two machines. We describe techniques later in this section which preserve this high degree of protection against hardware failures while also providing reliability and security. While both machines are in operation their results can be compared to detect hardware errors; however, once one machine has failed, continued operation is still possible if protection against security failures due to further hardware errors can be tolerated in this degraded mode. If this is unacceptable, secure and reliable operation can be achieved by adding another copy of the critical hardware. The incremental cost of adding security (triplication) to a reliable (duplicated) system is only on the order of 50% in the critical portion of the system, which in turn is a small part of the overall system.

5.2.3 Physical Separation

The system design that we recommend uses physical separation into three parts to achieve a high degree of security in a reasonably efficient manner. The three physical modules are referred to as blue, green, and orange. The blue and green modules (executing the same algorithms) together act as the security monitor. They perform tasks where hardware or software failures are critical. The orange portion performs the bulk of the operations, but calls on the blue and green portions to perform critical parts. This system's structure is illustrated in Figure 13.

In addition to the three basic modules the figure illustrates the I/O, which is connected to the blue and green modules. There is a communications path between the blue and orange modules and another one between the green and orange modules but there is no communication directly from the blue module to the green module.

The key concept that makes this approach work is a mechanism which assures that the operations performed by the orange machine are in fact not critical to the security of the system. This mechanism is the checksummed data unit which is described in the next section.

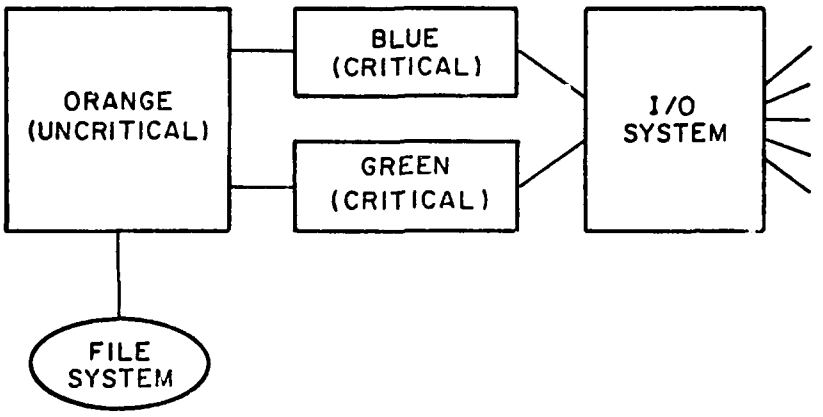


Figure 13 System Physical Separation

5.2.4 Secure I/O Structure

Secure operation of the I/O system will be assured by requiring concurrence by blue and green on every system input and output. Two methods of obtaining this concurrence are:

1. dual port interfaces with a port for blue and another one for green; and

2. checksums and port identification numbers on all segments.

These techniques were discussed previously in section 5.1.1 where we mentioned the security hazards in the I/O system. These two techniques are appropriate in different portions of the I/O system. The use of checksums and port identifications on all data segments is an appropriate technique for a high speed device which operates in a direct memory access mode. Since the amount of logic in the device which is dedicated to accessing memory is quite large, it is uneconomical to duplicate this logic in every device and provide duplicate paths to independent memories. On the other hand, dual port interfaces are quite appropriate for polled interfaces since the amount of logic associated with the I/O bus is quite small.

The use of checksums and port IDs in all segments naturally leads to the possibility of putting that type of I/O device on the orange machine provided that the format of the data packets permits the I/O device to check whether this segment has been authorized for output by blue and green (since the blue and green machines must check the security level and user name access field before a segment can be delivered to the user). This requires that the devices be able to test the blue and green checksums.

These techniques are not necessary for the I/O devices in the file system such as the disk and magnetic tape units, which transmit complete checksummed data segments including the checksum, since we assume that the disks and magnetic tape units are maintained in a physically secure environment along with the PMS computers.

The redundancy in the I/O system required to insure secure operation interacts with the redundancy required in the system for reliable operation. As we mentioned before, options exist where one could select reliably secure operation or, alternatively, operation which is reliable but is only secure when all system components are operational. Even in achieving the reliably secure system configuration, it is possible to modify the structure of the Pluribus I/O system to achieve reliability and security at a reasonably low cost.

Figure 14 illustrates a reliably secure configuration. The orange machine is configured as a reliable Pluribus with redundant processor and memory busses. A spare machine is included to execute the critical portions of the code should either the blue or green machine fail. Notice that simply making the blue and green machine reliable is an overly expensive solution since not only must the machines be duplicated but also the paths to both orange and the I/O system must be duplicated, which increases the cost of the critical portion of the system by 100%. The I/O system is shown with three ports because in order to assure security, two ports must be available even when one port has failed, so that a spare must be provided.

Figure 15 illustrates the structure of a polled I/O device in the system. It has three ports, one to each of the blue, green, and spare machines with independent address recognition and I/O interface logic for each of these parts. These three sets of I/O logic feed the logic which is dedicated to the data transaction of this particular I/O device. The path to each of the I/O busses includes a watchdog timer as well as the device address logic and data logic. During normal operations, blue, green, and spare machines access every I/O device periodically. The logic in each I/O device which is responsible for insuring

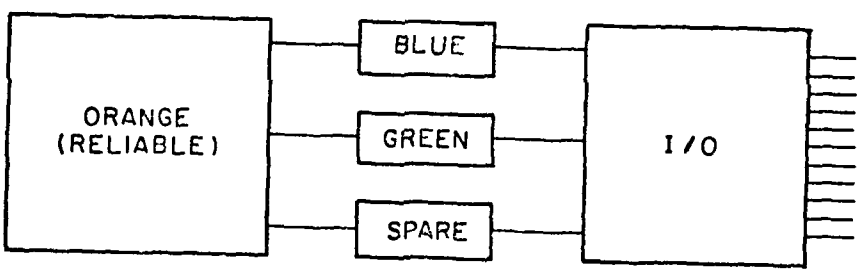


Figure 14 Reliable System Configuration

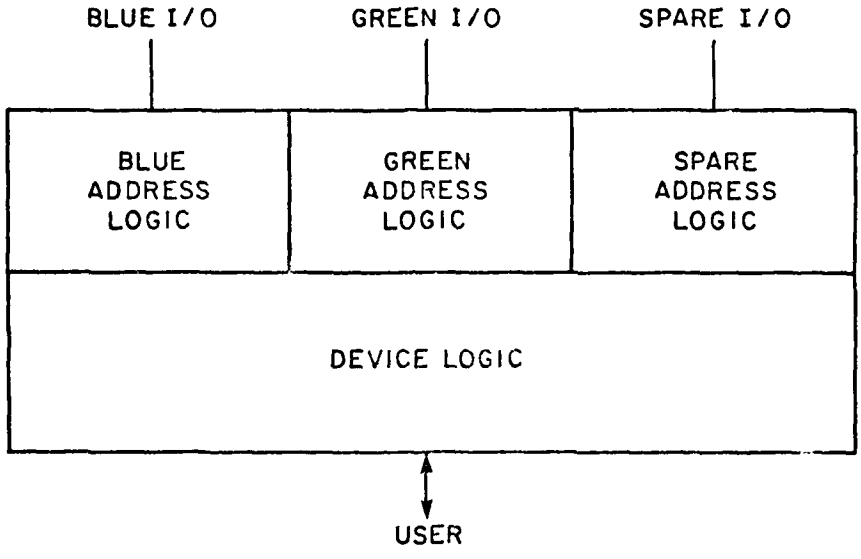


Figure 15 Polled I/O Device Structure

that at least two of the critical machines concur on every input or output operation uses these watchdog timers in the following way: If the watchdog timer for one of the I/O busses indicates that that I/O bus is active, then the data on that I/O bus must agree with every transaction that the device performs. If, however, the watchdog timer indicates that the I/O bus is idle (i.e., the machine on that I/O bus has failed), then concurrence on that I/O bus is not necessary before a transaction can be processed by the I/O device. This logic furthermore requires that at least two of the watchdog timers indicate activity, otherwise no I/O transactions will be performed by that device. This technique assures that concurrence from at least two machines will be required on every I/O transaction, yet only two of the three machines need be operational at one time. This is an attractive approach because it provides both security and reliability with only a small increase in the cost of the overall system.

5.2.5 Software Verification

As we have alluded to above, in Section 5.1.4 and in the introduction to Section 5.2, software verification will be required for some critical portions of the system. We have minimized the portions of the system requiring software

verification and grouped them together to be run on the physically separated green and blue portions of the hardware system. Thus, we have separated out a "kernel" of the system to be verified. This is in keeping with the current state-of-the-art of software verification for which "security kernels" are frequently used.

Using the common nomenclature, a security kernel is a software module used to augment existing hardware provisions for enforcing access controls within a computer system. The basic idea is that a correctly designed and implemented security kernel can enforce the security requirements for any unvalidated programs running on it. For example, at MITRE a security kernel has been designed on a PDP-11/45 and its correctness is being proved[29]. They use two steps to prove the correctness of a security kernel. The first step is to validate a formal specification of the program with respect to axioms for a secure system. The second step is to demonstrate or prove the correct implementation of the programs.

---

[29] J.K. Millen, "Security Kernel Validation in Practice", CACM, Vol. 19, No. 5, May 1976, pp. 243-250; W.L. Schiller, "The Design and Specification of a Security Kernel for the PDP-11/45", ESD-TR-75-69, the MITRE Corporation, Bedford, Mass., March 1975.

We will not go into further detail in this report about the techniques for software verification. We have recently submitted to ARPA a document[30] which demonstrates our understanding of this area. Suffice it to say that for the PMS we would use a mix of the pragmatic and the theoretic similar to that proposed in the above-mentioned document, maximizing the probability of obtaining a verified operational system, rather than becoming bogged down in abstractions or verifying a toy system.

Note that in the design of conventional security kernels, it is assumed that the hardware will work properly (i.e., all programs in a security kernel will be executed correctly by the hardware). No attempt is made to prevent security violations due to hardware failures. In the real world, the hardware will go down. For example, the contents of the data structures within a security kernel may be changed by a memory failure. Hardware failures definitely should be considered in a security kernel design (i.e., how to design a reliable security kernel), and we have done this.

---

[30] BBN Proposal P76-CSY-80, "Consolidation, Documentation, Certification, and Maintenance of the UNIX Operating System," pp. 22-56, Appendix I.

5.3 Conclusions

The methods described in this section combine to create a secure environment for message processing. The system is designed to protect against failures from both hardware and software. I/O is protected from failure by duplication and assists in failure detection by monitoring message checksums. Data interference is protected against by physical separation of the I/O equipment. Finally, message integrity is protected by separation of blue and green processes and by completed message checksums which are verified at each critical point.

We emphasize again that this design goes much further than anyone has ever gone before. Previous systems have been built using redundant hardware to protect against hardware failures. Previous systems have also used verified software to protect against software failures. To our knowledge, no one has previously addressed the issue of preventing failures when code, albeit verified, is run on hardware which can fail, the only truly realistic assumption. Further, this has all been done without inordinate amounts of costly hardware.

6. HIGH ORDER LANGUAGE

It is our opinion that the software for a PMS should be written in a high order language (HOL). The reasons for such a decision are well known and need merely be summarized here:

- Preparation efficiency and productivity increase using a HOL.
- HOL code is easier to read and therefore easier to modify than assembler code.
- HOL code is easier to maintain.
- If verification is to be done, it is practically possible only if a HOL is used.

Given these advantages for using a HOL, why is there any question about using one? Two objections stand out: existence of a compiler, and efficiency of the code. No existing compiler for any HOL produces code for the Pluribus. Thus, a decision to use a HOL for the PMS would require creating such a compiler. Efficiency of the compiled code then becomes an issue. The PMS will be a high bandwidth system, requiring truly efficient code. All previous Pluribus code has been written in assembly language. A compiler that produces really high quality code at the state of

the art in compiler production is quite expensive. Of course, use of a less efficient compiler can be compensated for in a Pluribus environment by using more hardware, so the issues involved in the tradeoff between hardware and software costs are relatively straightforward.

In spite of these objections, the possible payoffs appear to be large enough that we have investigated the matter further. A compiler for SUE would not run on Pluribus but would be a cross-compiler from another machine, such as TENEX. Since the SUE processor is similar in many ways to the PDP-11, we have investigated the possibility of modifying some existing compiler for the PDP-11 that runs on TENEX. Two obvious choices are BCPL and BLISS.

Two compilers for BCPL currently run on TENEX -- one for the PDP-10 and one for the PDP-11. The former has just gone through an extensive improvement process and now produces moderately good code. Although one might guess that a compiler for SUE could be most readily produced by modifying the PDP-11 compiler, our investigation has revealed that the PDP-10 compiler is a better starting point. In addition to the fact that the PDP-10 compiler is a better compiler, there is the problem that the PDP-11 compiler makes extensive use of the PDP-11 memory-to-memory

opcodes which are not available in the SUE. We estimate that in perhaps six person months we could modify the existing TENEX BCPL compiler for the PDP-10 so that it would produce moderately good SUE code.

The BLISS language was developed at Carnegie Mellon University as a system programming language for the PDP-10; BLIS11 is a BLISS compiler for the PDP-11 that runs on the PDP-10 and can be run under TENEX. The people at Carnegie have put a lot of effort into code optimization, so that BLIS11 produces very high quality code -- probably about as good as the average programmer, although a skilled programmer does better when he is trying hard. The code generation strategies used in BLIS11 turn out to be more easily adapted to the SUE than are those used in the BCPL compiler, so we estimate that in about three person months we might have a quite high quality BLISS compiler for the SUE, running on TENEX.

The verification efforts that might be part of this project suggest Euclid as a candidate, since it has been designed specifically to facilitate verification. Although this advantage cannot be ignored, Euclid has some serious problems that must be considered. It is not clear that the language design process is

at its end, since the latest document on Euclid[31] is marked as a draft and apparently is to be modified. So, although the language appears attractive in its present form, we have no way to know how it will look when finished. Moreover, there are some language issues in the present document that give us pause. The document states (page 2) that the "design does not specifically address the problems of constructing very large programs; we believe most of the programs written in Euclid will be smaller than about 2000 lines." It is difficult for us to evaluate the implications of this statement, although it seems likely that the code to be verified for the PMS will almost certainly be considerably larger, and the total system will be at least an order of magnitude larger. The document goes on to "assume that the user is willing ... to obtain verifiability by giving up some run-time efficiency, and by tolerating some inconvenience in writing his programs." Again, this is difficult to evaluate.

In addition to these potential language problems, there are some possible problems with the implementation. As mentioned above, the language design process is not yet complete. It follows that no compiler is yet available and implementation of

[31] "Euclid Report", B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchel, and G. J. Popek, 20 Apr 76, DRAFT for criticism.

one can hardly start before completion of language design. On the other hand, a Euclid compiler could probably be created by modifying an existing PASCAL compiler, not likely to be a large task. However, current PASCAL compilers usually require a runtime support package, which would be awkward at best in the Pluribus environment. Also, the PASCAL compiler for the PDP-10 does not produce particularly efficient code.

Euclid has too many unknowns for us to evaluate it properly. The compiler situation for BLIS11 is much better than that for BCPL, although BCPL has the advantage that all needed expertise is in house. As a language, we prefer BCPL to BLISS, although the differences are not of great significance. BLISS has no "goto" statement, sometimes a moderately serious lack. Further, one could not readily be added, since the code optimization strategies which make the compiler so attractive are heavily dependent on there being no "goto". Although we favor the use of some HOL to be used in implementing a PMS, the situation is presently sufficiently fluid that it is premature to select a language now; the decision should be made just before starting the implementation. However, if we had to make a choice now, we would select BCPL.

Ignoring temporarily the issue of which HOL to use, we have given some thought to what modifications might be needed in any HOL to adapt it to the somewhat unusual environment presented by Pluribus. For convenience, our thinking has been in terms of BCPL. We note the following problems that would have to be solved as part of an effort to produce a useful BCPL compiler for the Pluribus.

1. We must modify an existing compiler to produce SUE code.

2. The Pluribus uses map registers to permit a processor with a 16-bit address to access 2**20 bytes of memory. Programmers find that dealing with the map registers efficiently is a significant part of writing good Pluribus code. Ideally, the compiler would totally insulate the programmer from this task, but this might be too much to hope for. The proper answer is probably to provide the programmer with linguistic constructs suitable for advising the compiler about how best to handle map registers.

3. Life in a multi-processor environment requires certain special linguistic constructs, such as those to deal with interlocks, explicit parallel processing, etc.

4. Pluribus code is broken into short (in time) segments called strips, and in each Pluribus application there is the requirement that no strip execute for more than some number of milliseconds. The compiler may be able to help the programmer deal with strips, but it will probably be necessary for the programmer to advise the compiler.

5. Although there is no "operating system" as such in Pluribus, there is the reliability package that is part of all Pluribus systems. This is written in assembly language, and provisions will be needed to interface the HOL to it.

6. BCPL, BLISS and (apparently) Euclid require a stack at run time. It is not immediately obvious what this means in a multi-processor. Should there be a stack per processor? Or one per Pluribus PID level? Or a stack per user?

In addition to its capability of supporting a reliable and secure message system, Pluribus also offers the capability of high throughput through application of its multiple processors. Furthermore, as has been mentioned above, the task of message handling is in many ways a "natural" for a multi-processor, due to the parallelism inherent in the message-handling function.

This brings us to an important issue, however: to what extent do we expect parallelism to be discovered by the compiler, and to what extent do we expect the programmer to specify it? Our view is that the compiler will do very little sophisticated parallelism discovery. Automatic discovery of parallelism is a highly complex task which should not be placed in series with development of a message system. Instead, we expect that the programmer will maintain control over the parallel processing in the algorithm.

7. SYSTEM PERFORMANCE AND SIZING

In this section, we will investigate the configuration of an actual PMS. Since a Pluribus is so modular, we need to determine our requirements in many dimensions including processing power, memory size, and I/O. A convenient metric for message systems is the number of users, so we will attack this problem by determining the load placed on the system by a typical user.

There are, in general, two approaches to determining this load: analysis and analogy. When analysis is used, the various tasks performed in serving user requests are abstracted and estimates are made about 1) the processing time required to perform each task, 2) the memory space required to hold the program which performs these tasks and that required to hold the user data during execution of these tasks, and 3) the mass storage requirements associated with this aspect of the system. This approach can be accurate if there is a body of experience on which these estimates can be made. Furthermore, this approach is attractive when a system is being moved from one environment to another. On the other hand, the second approach to system sizing, analogy, considers as a whole the system to be developed and compares it to the performance of a similar existing system. Analogy has the advantage of providing answers to overall system

performance questions when it is not possible to give accurate estimates for the components of the system.

We have combined these two approaches to determine the configuration for PMSs. We have used measurements performed on an existing message system as an example. From these measurements, we have derived statistics on the average user activity (e.g., number of messages sent, number of addressees, time spent in other commands, etc.) and on the resulting performance of the system (e.g., CPU time to perform each task). This statistics gathering activity provides the analogy. We have extended it through analysis to determine from the user activities the required disk storage, network bandwidth, etc. The results of this study and of our analysis are described below, followed by a summary of the tradeoff between the number of users supported by the system and the cost of the system.

We faced a dilemma in trying to choose an analogous system to measure. As has been mentioned, there is a plethora of message systems in the world, with widely differing characteristics. By the very act of choosing one of them to look at in detail, we would be inviting our measurements to be biased by the prejudices and inefficiencies of that particular implementation.

But this need not be disastrous. Although they differ greatly in style, message systems really have many things in common. They all must perform the same basic functions and thus, to some degree of accuracy, will have similar processing and memory requirements.

Independent of the choice of message system, another inaccuracy arises with respect to the user population. The notion of the load attributable to a "user" depends greatly on whether that user spends his whole day at his terminal, or whether his mail activity is only peripheral to his primary job.

We did, however, have to choose a system and an environment to at least give a starting point for our calculations. We decided to use Hermes on TENEX, knowing full well that this would not be a perfect analogy to the system which would be ultimately developed for Pluribus. This choice has the advantages of proximity of the implementors and a good instrumentation package built into the system.

7.1 Measurement of an Existing System

The instrumentation of Hermes is implemented as follows. As execution of each command starts and ends, the name of the command, the elapsed real time, the elapsed CPU time, and other

selected data are recorded. Through this mechanism, the amount of CPU time required for the execution of each command was determined as well as the frequency of use of the various commands. The interval between the end of one command and the execution of the next command was also determined in this way.

Table 1 Measurements of Hermes

Total Sessions:	973
Avg. Real Time:	1422 sec.
Avg. CPU Time:	16.3 sec.
Total Messages:	572
Avg. Addressees per Message:	2.67
Avg. Length of a Message:	1234.7 chars.

Table 1 summarizes the results of this study. Over 900 sessions were measured over a period of about a week at several TENEX sites. The average session lasted 1422 seconds (about 24 minutes), during which time 16.3 seconds of processor time were used. Dividing these two numbers, we find the maximum number of users which could be using the system at the same time:

$$1422/16.3 = 87 \text{ users}$$

However, because of queuing limitations, the practical limit to the number of users will be much lower. Queuing effects seriously degrade operation of a system if it runs above about

80% of capacity for short periods of time. The calculation is then:

$$87 * .8 = 70 \text{ users}$$

Thus, a TENEX should be able to support about 70 active Hermes users. If the PDP-10 were programmed to run Hermes without the overhead of the operating system, and if Hermes were rewritten to increase its efficiency, this number could be increased significantly. However, the PDP-10 is not a multiprocessor, so it is not practical to increase the number of processors performing these tasks. One of the central advantages of the Pluribus is the ability to add resources to satisfy a higher load.

We have used the statistics about the frequency of user command usage to determine the amount of disk activity produced by a user action. These statistics are given in Table 2. This table gives a breakdown of the 973 sessions which were summarized in Table 1. It gives the average number of times the command is used per session, the average real elapsed time during execution of the command (in seconds), and the average CPU time consumed in execution of the command (in milliseconds). For example, the "REPLY" command was used on the average .222 times per session, and the average elapsed time from the termination of the command

Table 2 Command Statistics

COMMAND	USAGE	REAL TIME (SEC)	CPU TIME (MILLISEC)
Compose Messages:			
REPLY	.2219938	347.6111	4568.19
EDIT	.08119219	167.6709	1646.835
COMPO	.2291881	502.1031	4975.422
CREAT	.2045221	426.1307	5010.789
SEND	.01336074	16.38462	1220.0
MAILE	.09660843	335.9681	1444.234
FORWA	.1181912	341.1826	3991.191
ERASE	.02055498	13.0	221.75
SHOW	.08941418	18.05747	739.6092
SUGGE	.004110997	622.0	7378.25
EXPLO	.006166495	331.5	5449.333
Look at Messages:			
PRINT	.7790339	92.61609	2593.319
S	.2127441	21.19807	1898.894
GET	.3597122	18.46	2440.837
UPAR	.03494347	26.44118	1314.412
LF	.7800617	24.16733	1149.494
SURVE	.393628	36.03916	2872.624
TRANS	.01849949	18.22222	3536.222
CONSI	.02363823	11.95652	869.7826
LIST	.09249743	24.56667	5452.856
SUMMA	.05035971	23.87755	3115.49
JUMP	.002055498	3.0	101.5
Manipulate Messages:			
DELET	.4655704	6.320088	286.5607
FILE	.3627955	19.20397	978.6912
UNDEL	.01849949	3.222222	275.0556
D	.07194245	4.428571	283.8714
MOVE	.1130524	20.56364	1130.209

Hermes Houskeeping:

EXPUN	.04110997	8.6	2221.75
QUIT	.4316547	5.364286	565.7286
BSYS	.04316547	19.69048	1964.31
Q	.5477903	5.245779	553.2176
STATU	.1212744	1.771186	177.6949
CNTO	.09146968	24.62921	537.5169
MARK	.03597122	8.0	690.5143
EXPOR	.01336074	14.84615	601.5385
EXIT	.02466598	26.125	4271.5
COPY	.004110997	12.75	428.5
USE	.01130524	11.09091	220.1818
IMPOR	.002055498	9.0	418.5

Miscellaneous:

EXEC	.1336074	1427.923	724.7692
MAILS	.02980473	16.68966	1098.483
VERSI	.006166495	24.83333	986.3333
RETRI	.006166495	8.0	124.5
CHECK	.006166495	8.0	666.3333
DESCR	.05960946	38.67241	1155.603
NEWS	.02158273	42.47619	1012.619
JOBST	.002055498	3.5	754.5
EXPLA	.004110997	338.0	4109.25
DIREC	.03494347	21.08824	1162.794
HELP	.01233299	466.75	7185.917
DAYTI	.005138746	5.0	97.6
SEMI	.002055498	6.0	116.0
TALK	.001027749	309.0	431.0

line to the return of the prompt for the next command was 347.6 seconds. The average CPU time consumed during execution of the "REPLY" command was 4568 milliseconds. The elapsed real time for some commands is quite large because it includes the time it took for the user to fill in answers to various subcommands.

The messages are grouped into those which create or compose messages, those which examine messages, those which manipulate or file messages, those which perform internal tasks, and those which provide miscellaneous features (such as the time of day). From these statistics, we have determined the load produced by an "average" user.

7.2 Processor, Memory, and Disk Requirements

Understanding the necessary inaccuracies of our procedure, we can now extend the results derived in the previous section from an existing system to the proposed Pluribus architecture in order to determine the processing, memory, and disk requirements. Our goal is a system which will support 1000 users.

As we calculated earlier, one TENEX processor can support 70 users. As a result, a TENEX-based message system which supported 1000 users would require 14 PDP-10 processors. A Pluribus processor is about half as fast as a TENEX processor. However, we feel that the TENEX processor is being used at less than full efficiency for Hermes for the following two reasons:

- TENEX is an extremely powerful operating system, but much of its generality is not being used by Hermes.

- Hermes itself was not written with run-time efficiency as a goal. As a result it is much less efficient than it could be.

We might therefore estimate that the PDP-10 processor is about 50% efficient for Hermes, and come to the (somewhat surprising) conclusion that a Pluribus processor would about equal a PDP-10 processor for this application. As a result, we estimate that one Pluribus processor will serve about 70 users. Thus a 14-processor Pluribus system would be necessary to support 1000 users.

The requirements for disk space can be divided into two classes: message file storage and working storage. The volume of message file storage dominates the volume of working storage to such a great extent that it is safe to ignore the latter. Our results from the Hermes study (see Table 1) showed that during 973 sessions, 572 messages were created. The average session lasted about 24 minutes. This corresponds to an average message generation rate of 1.5 messages per user per hour, or 12 messages per day. At an average message size of 1234 characters and assuming 20% overhead, a user generates 17,800 characters of message storage per day. If we assume that these numbers are valid for the future Pluribus environment, then we can compute

the amount of disk storage required as a function of the number of users and the length of time messages are retained by the formula:

Disk Size (megabytes) = Days * Users * .0178

For our 1000 user model, the disk storage requirement varies as follows:

Storage (Days)	Disk Size
10	178 megabytes
20	356 megabytes
30	534 megabytes
60	1000 megabytes

Since the largest disk drives conveniently available have a 300 megabyte capacity, multiple drives are required for long term storage. Thus, the price of disk storage is a direct function of storage time.

There is a further question of whether the bandwidth to and from a disk is adequate to support the load of 1000 users. In this case, as in the case of disk volume, there are two types of load: message storage and retrieval and temporary storage (like swapping). From an analysis of the Hermes statistics, we have

found that the load on the disk due to message storage and retrieval is about 7.5 characters per second per user. Thus, 1000 users require 7500 characters per second, a figure well within the capability of these disks. The requirements for temporary disk access are both greater and much harder to estimate. Indeed, the swapping behavior of a multiprocessor multi-user system is outside the scope of this study. Actual determination of this parameter can be made after an initial implementation. If it turns out that a 3330 type disk cannot support this access rate (which is quite likely), there are at least two alternatives: drums (or fixed head disks) and EBAMs. Since the access time of EBAMs is about three orders of magnitude faster than disks, we feel comfortable that the short term storage needs of even a 1000 user system can be met by an EBAM device. A smaller system may be adequately supported by a disk alone.

The main memory requirements of the message system are also divisible into two types: program and system variables and user storage. We again begin by analogy to an existing system. The Hermes memory requirements are summarized in Table 3.

Table 3 Hermes Memory Requirements

Pure Code:	77300 words
Library Routines:	7000 words
Auxiliary:	12000 words
Total:	96300 words
Per User Storage:	7500 words

The TENEX word is 36 bits long while the Pluribus word is only 16 bits. To compensate for this difference, we estimate that a TENEX word is equivalent to two Pluribus words. As a result, the code requirement to implement a similar message system on the Pluribus would be 200,000 Pluribus words.

We believe that these code requirements are low enough that it would make sense to provide enough memory to keep code in memory all of the time, rather than resorting to complex techniques of overlays or code swapping. Furthermore, in a multiprocessor environment it could prove to be difficult to implement these mechanisms. By keeping the code in memory, we avoid these problems at a reasonable cost, and achieve efficient operation.

We believe that in an optimized implementation, the amount of memory necessary to support an individual user can be much less than implied by our analogy, perhaps on the order of 100 words per user always resident in memory plus a few thousand words of working storage swapped among the active users.

7.3 Terminal Access

Up to now, we have been discussing the PMS as though it were a stand-alone host connected to a communications network, but without terminals of its own. As mentioned in Section 3.1, another attractive approach is to integrate the PMS with a Pluribus TIP to support local users. If this were done, the number of processors and the amount of memory needed would increase. The Pluribus TIP code requires about 6,000 words of memory plus 40 words per terminal. The processor requirement for the Pluribus TIP is .01 processors per terminal plus one processor for background and periodic tasks. Thus, a 1000 user PMS requires an additional 46,000 words of memory, and 11 extra processors to support the terminals.

In addition to the processor and memory requirements, terminal support obviously requires termination points for the terminals. The Pluribus TIP uses the Multi-Line Controller which

was developed for the Honeywell 316 TIP. One MLC can support up to 63 terminals. As a result, a 1000 user system with all local users would require 16 MLCs. (Such an extremely concentrated environment would also probably require other termination equipment such as a tech control installation.)

7.4 Pluribus Configurations Summary

Throughout this section, we have been assuming a configuration that would support 1000 users. Let us now summarize that system, assuming that the users would access the PMS via a network. This system would have these characteristics:

- 17 processors
- 10 processor busses
- 2 300-megabyte disc drives
- 2 tape drives
- 400,000 words of memory
- 3 memory I/O busses
- doubled interfaces to the disks, tapes and network
- 1 4-megaword EBAM

This configuration can take advantage of the ability to combine memory and I/O busses into one logical bus to reduce the number of bus couplers. The system described above has enough

redundancy to survive any failure, including entire busses, without significant degradation. It also includes the three small "sub-machines" to perform the blue and green security functions (one is a spare).

This system is exemplary of a large PMS which would be used in an actual operational environment. We can of course take advantage of the inherent modularity of the Pluribus to configure a system for whatever characteristics a particular environment might have.

In any event, the system as described is much larger than would be needed for the initial development of the PMS software and security techniques. We have therefore included a summary of another smaller configuration which might be appropriate for development. This machine could later be expanded to a larger operational system, if desired. The system is configured to support about 100 users and has provision for both local and network terminal connections. Its characteristics:

- 7 processors
- 5 processor busses
- 100 megabytes of disk memory
- 1 Multi-Line Controller

- 1 tape drive
- 250,000 words of memory
- 2 memory I/O busses
- doubled interface to the network
- doubled interface to the tapes and disk

Once again, the system can be configured to take advantage of combined memory and I/O busses to reduce the cost of the system. This system has one processor for each of the blue, green, and spare sub-machines and four for orange. This system has less redundancy in the processor system than the previous one, but because two processors can support 140 users, even with one bus out of service, there is still enough orange processing power to support the users. The memory is not configured to be fully redundant, so the system cannot survive the loss of a memory bus as a whole, although it can proceed if any memory element is defective.

Appendix A The Pluribus "Stage" System

The claim is made in the body of this report that the Pluribus architecture permits programs to be written in a manner that affords high reliability. To help the reader to understand this claim, we present in this appendix a more detailed description of the Pluribus reliability package and an example of its use in one particular Pluribus application -- the IMP. This appendix assumes detailed knowledge of Pluribus architecture; see, e.g., [32].

The "stage" subsystem has the job of keeping track of just which resources, both hardware and software, are available in a Pluribus system, and coordinating their use. An application built on the stage system, be it an IMP, a CCP or whatever, would execute in a virtual environment where a (working) subset of the available machine resources is set up for the use of the system and the remaining resources are either kept in reserve as spares or are being held aside pending repairs. If a resource in use fails, the stage subsystem will automatically switch a spare into the system's active environment, allowing the system to continue with at most the need to reinitialize some portion of itself.

---

[32] Pluribus Document 2, "System Handbook," BBN Report 2930, January 1975.

The stage subsystem is organized as a series of hierarchical routines (or stages) which all of the available processors run asynchronously and in parallel. Each stage determines just what resources are available in some portion of the system's environment and selects some of these resources for use by the system. A processor may not proceed on to the execution of the next stage in the hierarchy until it has reached agreement with all of the other processors, and the application system is not run until all of the stages have been executed. Even after a processor begins running the application system, it continues to regularly run the checks of each stage incrementally and if a change in the environment is detected by this process the processor "traps" back to the appropriate stage, makes any necessary changes to the system's virtual environment and then works up through the later stages to resume running the application system. If the change is minor the system will continue, having experienced only a short "hiccough," while if the change is significant the system will have to reinitialize itself.

The basic mechanism by which the processors reach agreement in each stage is called a "consensus". Each function of each stage has a separate consensus word. As a processor begins

performing any particular stage function, it "checks in" to the appropriate consensus by setting in the consensus word a bit representing the processor. Thus, each consensus word always contains the bits of all the processors participating in that particular function. If a processor detects an anomaly, it set its bit in a "fixit-word". If the fixit-word and the consensus agree, then the repair, update, table change, or whatever, is made and the fixit-word cleared; if the fixit-word and the consensus do not agree, the processor hangs at that point in the stage system waiting either for itself to be repaired, in which case it will no longer desire the fix and will continue through the stages, or for all the other processors to notice the anomaly, in which case the fixit-word will eventually agree with the consensus, the fix will be made, and all the processors will be freed to continue through the stages.

For example, suppose a processor loses its coupler to one of the memory busses. It will eventually notice that a portion of memory is missing and will check into the appropriate fixit-word to have the system's available memory tables updated. However, none of the other processors will see any anomaly, and hence the fix will never be allowed. The processor will be stuck at that point in its stage subsystem until its coupler is repaired. The

processor's own bit will be timed out of the higher consensuses, the processor will not run the application system, and the processor will have effectively isolated itself from causing any trouble to the properly functioning processors. Now suppose instead that the memory bus itself develops some problem. For each processor the scenario proceeds as above (in fact, the individual processor cannot tell the difference between this case and the last!), but in this case all the processors will see that some memory is missing and hence they will all check into the appropriate fixit-word. Thus, the system's available memory tables will be updated and all the processors will resume running the system. Notice that in both cases, precisely and only the failing component was removed from the system, which otherwise continued to run.

In the IMP application, the code is divided into 9 separate asynchronous stages, numbered from 0 to 8. The system periodically executes each of these components. Each processor executes them and as soon as a test fails, that processor will only execute tests with number less than or equal to the number of the failed test. Each test has common exit conventions; there is a good exit and bad exit from each test. The bad exit causes the execution only of tests with number less than or equal to current number.

Stage 0 sets up the interrupt vectors for the processor, both the interrupt vectors for devices and the interrupt vectors for quits and illegal instructions. It then checksums the local memory so that future changes to local memory can be found. The stage fails if the checksum does not check with the previous checksum.

Stage 1 finds those pages of common memory that are available. It executes a test on each of the pages and builds a table using the constant table of all available pages. All active processors must agree before an entry can be changed in the table. The page on which all processors will communicate is chosen, and its map is stored in a register.

In Stage 2, processors discover who they are and what set of processors exist in the machine.

Stage 3 recomputes the checksum of common memory pages.

Stage 4 reconfigures common memory, assigning various pages to the various tasks. It attempts to keep variables and buffers pages off the code bus and to keep copies of pages on separate busses. It makes copies when they are needed.

Stages 0 through 4 reside in local memory for each processor since it is not known which pages of common memory are available. After stage 4, this is known, so stages 5 and up reside in common memory code page three.

Stage 5 performs initialization of buffers and variable pages as needed. The successful operation of the system holds timers which inhibit any initialization from happening.

Stage 6 checks the I/O devices to see which exit. The check made is a simple read of a device register to see if a quit occurs. Existing devices are noted in a table set up for the configuration module. This stage also chooses which PIDs are usable and which clock will be used. Variables are set to the PID numbers of this clock so that clock-driven modules will start only on the clock in use.

Stage 7 finds which processors are usable and keeps them running. Each processor checks to see if the other processor on its bus should be started. All processors cooperate to start processors on separate busses.

Stage 8 performs some checks which must be done by each processor.

The application programs reside at stages 9 and above.

Appendix B Description of the Pluribus

This report describes a proposed implementation of a message handling facility on BBN's Pluribus hardware. Although we recognize that most readers in the ARPA office are familiar with this hardware and the motives that led to its design, for the sake of completeness and for the benefit of non-ARPA readers we feel it appropriate to discuss these matters in this chapter. We present the motivational issues in the remainder of this section, the Pluribus hardware in Section B.1, and issues relating to Pluribus software in Section B.2.

The Pluribus computer is a modular multiprocessor based on a commercial minicomputer. Its architecture is an outgrowth of the new and flexible computer structures which began to appear in the early '70s. The primary goals of the Pluribus are flexibility (i.e., the ability to expand or contract smoothly over wide ranges) and reliability. Originally, high throughput was considered a primary goal, but this was soon seen to be balanced by the need for a cheaper, smaller machine with low throughput. Bandwidth is thus one of several domains in which flexibility is desired.

Let us consider the issue of flexibility in a little more depth. In most machines certain hardware "utilities" are shared among the various logical units. These include rack space, power, cooling, etc. Generally these utilities come in fairly large units, with correspondingly large steps in cost. Thus, one can typically add, say, interfaces up to a certain point; at which time a new rack, power supply, etc., must be added to permit further expansion. Even then, one may run out of logical channels or come up against other hard boundaries. The Honeywell 516/316, for example, has a fixed memory channel arrangement which limits connection to a total of at most seven high-speed circuits and/or Host computers. The specific component which is totally inflexible in most systems is the processor; that is, there is typically no processor modularity or possible variation of processor capacity. The flexibility goal of the Pluribus was to smooth large step functions in cost by utilizing a highly modular design and to push really hard boundaries (such as absolute limits on memory addressing capabilities or processing capacity) well beyond requirements anticipated at least for the next few years.

Now consider reliability. If a single computer fails on an average of ten times a year, then a collection of ten computers,

treated as a unit, will fail an average of 100 times a year. However, suppose that rather than viewing the ten computers as a unit which is down if any one of its constituent computers is down, we view the ten computers as a unit which is up as long as any of its constituent computers is up. If the mean time to repair a failed computer is small compared to the time between failures, the probability that all ten computers will all be down is very small. The reliability of the Pluribus takes advantage of such probabilities[33].

With the goals of flexibility and reliability in mind and with the price and size of minicomputers dropping, it was decided that the Pluribus should be built along the lines of a minicomputer-multiprocessor, or more generally, a multi-resource (processors, memories, I/O channels, etc.) system.

In considering which minicomputer might be most easily adaptable to a multi-resource structure, the internal communication between the processor and its memory was of primary concern. Several years ago machines wer introduced which

[33] Note that a key assumption is that individual failures are independent of one another. Although it is impossible to guarantee this independence (flood, total power failure, sabotage, etc.), nonetheless considerable attention has been given in the Pluribus design to maintaining as much isolation as practical so that one failure does not induce another.

combined memory and I/O busses into a single bus. As part of this step, registers within the devices (pointers, status and control registers, and the like) were made to look like memory cells so that they and the memory could be referenced in a homogeneous manner. One of the important features of this structure is that it made memory accessing "public"; the interface to the memory had to become asynchronous, cleanly isolable electrically and mechanically, and well documented and stable. A characteristic of this architecture is that all references between users are time-multiplexed onto a single bus. Conflicts for bus usage therefore establish an ultimate upper bound on overall performance, and attempts to speed up the bus eventually run into serious problems in arbitration. This structure forms a very clear and attractive architecture in which any unit can bid to become master of the bus in order to communicate with any other desired unit.

In 1972 a new computer was introduced -- the Lockheed SUE -- which follows the single bus philosophy but carries it an important step further by removing the bus arbitration logic to a module separate from the processor. This step permits one to consider configurations embodying multiple processors, as well as multiple memories and I/O, on a single bus. It also permits

busses which do not include any processor at all. The processor used in the SUE computer is a compact, relatively inexpensive (approximately $600 in quantity), slow processor with a microcoded inner structure. Table B-1 shows some of its characteristics. Its slowness and cheapness, of course, go together and since in a modular multi-processor, increased bandwidth is achieved merely by adding more processors, the weak/cheap processor has the advantage of allowing smaller steps

Table B-1 SUE Computer Characteristics

16-bit word

8 general registers

3.7 microseconds add or load time

Microcoded

Two words/instruction typical

8-1/2" x 19" x 18" chassis

64K bytes addressable by a single instruction

200 ns minimum bus cycle time

850 ns memory cycle time

425 ns memory access time

to be taken along the cost/performance curve.

In this section we have briefly described the technology changes which make possible a minicomputer multiprocessor. Reference [34] presents a more detailed discussion of this evolution. In the next section we discuss one particular minicomputer multiprocessor, the Pluribus.

B.1 Hardware Structure of the Pluribus

Several components of the SUE computer, mentioned above, were adopted for the Pluribus system, in particular the physical and logical bus, the processor, and the bus arbitration logic[35].

The hardware consists of asynchronous and independently functioning communication busses, coupled together. From a physical point of view, the SUE chassis represents the basic construction unit; it incorporates a printed circuit back plane into which 24 cards may be plugged. From a logical point of view this chassis includes a bus which provides a common connection

---

[34] Barker, W., "A Multiprocessor Design", BBN Report No. 3126, October 1975.
[35] For further details, see F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings 42, June 1973, pp. 529-537.

among all units plugged into the chassis. All specially designed cards as well as all Lockheed-provided modules plug into these bus chassis. With this hardware, the terms "bus" and "chassis" are used somewhat interchangeably, but we will commonly call this standard building unit a "bus." Each bus requires one card which performs arbitration. A bus can be logically extended (via a bus extender unit) to a second chassis if additional card space is required; in such a case, a single bus arbiter controls access to the entire extended bus.

One can build a small multiprocessor just by plugging several processors and memories (and I/O) into a single bus. For larger systems one quickly exceeds the bandwidth capability of a single bus and is forced to multi-bus architecture, shown in Figure B-1.

The functional units of the system (processors, memories, I/O controllers, and special devices) are distributed on these busses in such a way that units which must communicate frequently are placed on the same bus, whereas units which communicate less frequently will in general be on different busses. Units on the same bus can thus communicate at high speed without disturbing the remainder of the system. When a unit on one bus must communicate with a unit on another bus, some interference occurs

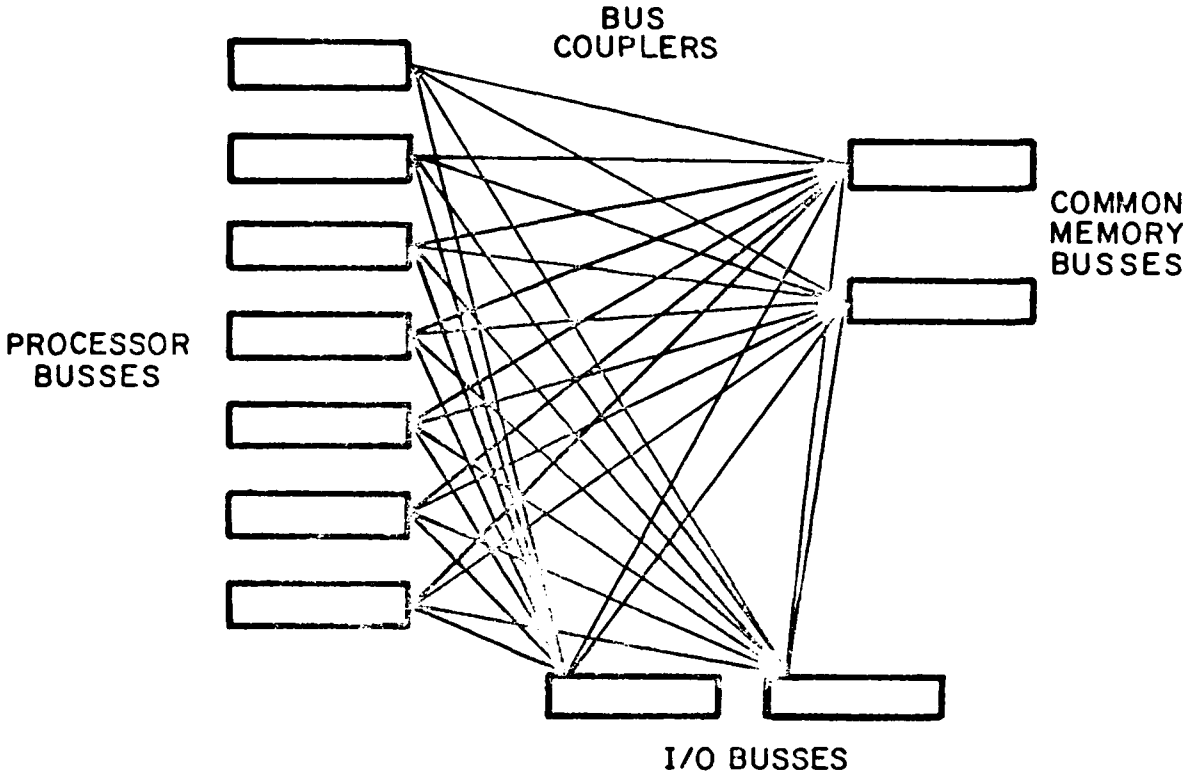


Figure B-1 Pluribus Structure

while both busses are momentarily involved in the interaction. Each bus, together with its own power supply and cooling, is mounted in its own modular unit, permitting flexible variation in the size and structure of systems. There are processor busses, memory busses and I/O busses.

This design is highly modular and permits systems of widely varying size and performance.

B.1.1 Resources

A central notion in a parallel system is the idea of a "resource", which we define to mean a part of the system needed by more than one of the parallel users and therefore a possible source of contention. The three basic hardware resources are the memories, the I/O, and the processors. It is useful to consider the memories, furthermore, as a collection of resources of quite different character: a program, queues and variables of a global nature, local variables, and large areas of buffer storage. The multiprocessor system is therefore in reality a multi-resource system, as mentioned above. The basic idea is to provide multiple copies of every resource so that the algorithm can run faster by using them in parallel. It can also survive a failure of an instance of a resource since other copies will be

available. The number of copies of the resource which are required to allow concurrent operation is determined by the speed of the resource and the frequency with which it is used.

It may seem peculiar to think of a processor as a resource rather than the controlling device, but in fact in a Pluribus system the parallel parts of an algorithm compete with each other for a processor on which to run. Indeed, a novel feature of the Pluribus design is the consistent treatment of all processors as equal units, both in the hardware and in the software. There is no specialization of processors for particular system functions, and no assignment of priority among the processors, such as designating one as master. Not only the application job but also the multi-processor control and reliability jobs are distributed among the processors so that all jobs are uniformly treated. The processors are viewed as a resource used to advance the algorithm; the identity of the processor performing a particular task is of no importance. Programs are written as for a single processor except that the algorithm includes interlocks necessary to insure multiprocessor sequentiality when required. The software thus consists of a single conventional program run by all processors.

B.1.2 Processor Busses

While the bus used in the Pluribus can support up to four processors, as this number is approached, contention for the bus increases, and the performance increment per processor drops. Pluribus systems use at most two processors per bus, which loses almost nothing in processor performance. When a processor makes access to shared memory via the switching arrangement, that access incurs delays due to contention and delays introduced by the intervening switch. In a typical application, some parts of the program are run very frequently and other parts are run far less frequently. This allows a significant advantage to be gained by the use of private memory. An 8K local memory containing an individual copy of the frequently run code is associated with each processor on its bus. This allows faster access to this "hot" code; the local memories all contain the same code. In the IMP application, for example, the ratio of accesses to local versus shared memory is better than three to one.

B.1.3 Shared Memory Busses

The shared memory contains program, buffers, global variables, etc. Buffer requirements, of course, vary depending

on the application. Since the bus is considerably faster than the memories, two logical memory units may be placed on each shared memory bus with almost no interference, to reduce contention between processors as they access this memory, and to increase the available storage.

B.1.4 I/O Busses

The I/O system consists of more standard busses. Into these busses are plugged cards for each of the various types of I/O interfaces that are required, including interfaces for modems, terminals, network connection, disks, etc., as well as interfaces for standard peripherals. The I/O bus also houses a number of special units including, for instance, (1) a clock; (2) a checksum/block transfer card which flows a block of memory through itself computing a checksum as it goes (used to checksum critical code, to compute end-to-end-checksums, etc.); and (3) a special hardware task disbursing unit known as a Pseudo-Interrupt Device (PID), discussed further below.

B.1.5 Interconnection System

To adhere to the requirement that all processors must be equal and able to perform any system task, busses must be connected so that every processor can access every part of shared memory, so that I/O can be fed to and from shared memory, and so that any of the processors may control the operation and sense the status of any I/O unit.

A distributed inter-communication scheme was chosen in the interest of expandability, reliability, and design simplicity. The kernel of this scheme is called a bus coupler, and consists of two cards and an interconnecting cable. In making connections between processors and shared memory, one card plugs into a shared memory bus, the other into a processor bus. Similar connections are made for every processor bus to every shared memory bus. When the processor requests a cycle within the address range which the bus coupler recognizes, a request is sent down the cable to the memory end, which then starts contending for the shared memory bus. When selected, it requests the desired cycle of the shared memory. The memory returns the desired information to the bus coupler, which then provides it to the requesting processor, which, except for an additional delay, does not know that the memory was not on its own bus.

The bus coupler also does address mapping. Since a processor can address only 64K bytes (16-bit address), and since we wished to permit multiprocessor configurations with up to 1024K bytes (20-bit address) of shared memory, a mechanism for address expansion is required. The bus coupler provides four independent 8K-byte windows into shared memory. The processor can load registers in the bus coupler which provide the high-order bits of the shared memory address for each of the four windows.

Given a bus coupler connecting each processor bus to each shared-memory bus, all processors can access all shared memory. I/O devices which do direct memory transfers must also access these shared memories. These I/O devices are plugged into as many I/O busses as are required to handle the bandwidth involved, and bus couplers then connect each I/O bus to each memory bus. Similarly, I/O devices also need to respond to processor requests for action or information; in this regard, the I/O devices act like memories and bus couplers are again used to connect each processor bus to each I/O bus. The path between processor busses and I/O busses is also used to allow processors to examine and control other processors for startup and trouble situations.

B.2 Software Structure[36]

The problem of building a message system lends itself especially well to parallel solution since messages can be treated independently of one another. Other functions of the program such as general housekeeping, routine computations, reliability tasks, etc., can also be easily performed in parallel. A software structure must be developed to exploit this parallelism. The structure chosen for the Pluribus works as follows: First, the program is divided into small pieces, called strips, each of which handles a particular aspect of the job. When a particular task needs to be performed, for instance upon receipt of a message over a communications circuit, the name (number) of the appropriate strip is put on a queue of tasks to be run. Each processor, when it is not running a strip, repeatedly checks this queue. When a strip number appears on the queue, the next available processor will take it off the queue and execute the corresponding strip. The program is broken into

[36] Further details may be found in R.D. Bressler, M.F. Kraley, and A. Michel, "Pluribus: a Multiprocessor for Communications Networks," Fourteenth Annual ACM/NBS Technical Symposium -- Computing in the mid-70's: an Assessment, June 1975, pp. 13-19. See also S. M. Ornstein, W. R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and F. E. Heart, "Pluribus -- a Reliable Multiprocessor", AFIPS Conference Proceedings 44, May 1975, pp. 551-559.

strips in such a way that a minimum of context saving is necessary.

Strips have different levels of importance. Data coming in over a high-speed communication circuit must be serviced more rapidly than data coming in over a Teletype-speed line. The number assigned to each strip reflects the priority of the task it performs. When a processor checks the task assignment queue, it takes the highest priority job then available. Since all processors access this queue frequently, the contention for it is very high. For efficiency, therefore, a special hardware device, the Pseudo Interrupt Device, was designed to serve as a task queue. A single instruction allows the highest priority task to be fetched and removed from the queue. Another instruction allows a new task to be put onto the queue. All contention is arbitrated by standard bus logic hardware.

The length of strips is governed by how long priority tasks can wait if all the processors are busy. The worst case arises when all processors have just begun the longest strip. In the IMP application, the most urgent tasks can afford to wait a maximum of 400 microseconds. Therefore, strips must not be longer than that. (Of course, a strip might be longer if it is run infrequently and if the urgent tasks do not have absolute

time requirements. That is, one might build a statistically acceptable set of strip lengths.) In the message system application, the interface requirements are more relaxed, and longer strips are acceptable.

An inherent part of multiprocessor operation is locking critical resources. This is the mechanism by which the algorithm enforces sequentiality when it is needed. Our system uses an indivisible load-and-clear operation (load an accumulator with the contents of a memory location and then clear the location) as its primitive locking facility (i.e., as the necessary multiprocessor lock equivalent to Dijkstra semaphores)[37]. To avoid deadlocks, we assign a priority ordering to our resources and arrange that the software not lock one resource when it has already locked another of lower or equal priority.

[37] E. W. Dijkstra, "Cooperating Sequential Processes", in Programming Languages, ed. F. Genuys, Academic Press, London and New York 1968, pp. 43-112.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD A104 052	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Interface Message Processors for the ARPA Network, QTR No. 6; and Final Report, Contract Line Item 0001AC	5. TYPE OF REPORT & PERIOD COVERED QTR: 4/1/76-6/30/76 Final: Line Item 0001AC
	6. PERFORMING ORG. REPORT NUMBER 3339
7. AUTHOR(s) F. Heart	8. CONTRACT OR GRANT NUMBER(s) F08606-75-C-0032
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, MA 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2351 Program Element Codes 62301E, 62706E, 62708E
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE July 1976
	13. NUMBER OF PAGES 185
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Range Measurements Laboratory Building 981 Patrick Air Force Base Florida 32925	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution Unlimited	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Computers and Communication, Store and Forward Communication, ARPANET, Packets, Packet switching, message switching, Interface Message Processor, IMP, Pluribus	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This quarterly Technical Report provides a final report on the suitability of the Pluribus computer to provide a message handling service of large capacity and high reliability while meeting stringent security requirements.	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)